October 1981

Report. No. STAN-CS-81-885

Also numbered:
HPP-81-16

The Emycin Manual

by

W. Van Melle
A. C. Scott
J. S. Bennett
M. Peairs

Department of Computer Science

Stanford University
Stanford, CA 94305

The EMYCIN Manual

William van Melle, A. Carlisle Scott
James S. Bennett, and Mark A. S. Peairs

9 November 1981

Abstract

This manual describes a domain-independent system, called EMYCIN, for constructing one class of expert computer programs: rule-based consultants. The resulting programs use knowledge specific to a problem domain to provide consultative advice to a client. The system-building tool, EMYCIN, is based on the domain-independent core of the MYCIN program. Domain knowledge is represented in EMYCIN systems primarily as production rules, which are applied by a goal-directed backward-chaining control structure. Rules and consultation data may have associated measures of certainty, and incomplete data entry is allowed. The system includes an explanation facility that can display the line of reasoning followed by the consultation program, and answer questions from the client about the contents of its knowledge base.

To aid the system designer in producing a knowledge base for a domain quickly and accurately, EMYCIN provides the following features: (1) a terse, stylized, but easily understood language for writing rules; (2) extensive checks to catch common user errors, such as misspellings; and (3) methods for handling all necessary bookkeeping chores. Other human-engineering features allow the system architect to produce, with a minimum of effort, a consultation program that is pleasing in appearance to the client.

Several consultation programs have been developed using EMYCIN, including consultants for medical problems and a consultant for structural analysis.

Copyright (c) 1981 The Board of Trustees of Leland Stanford Junior University. All Rights Reserved.

Research supported in part by NSF grant MCS-7903753

Table of Contents

1. Introduction 3

2. Contexts 15

2.1. The Context Tree 15
2.2. Uses of the Context Tree 18
2.3. Defining Contexts 21
2.4. Parameter Groups and Rule Groups 22
2.5. Properties of Context Types 22

3. Parameters 27

3.1. Defining Parameters 30
3.2. Properties of Parameters 31
3.3. Examples of Parameters 36

4. Rules 39

4.1. The Structure and Definition of Rules 39
4.2. The Rule Interpreter 41
4.3. Rule Groups 44
4.4. The Uses of Rules 46
4.5. Abbreviated Rule Language (ARL) 51

5. Building an EMYCIN Consultant 55

5.1. Organizing your Knowledge Base 55
5.2. Example: Acquiring a Business Consultant 57

6. Creating and Maintaining the Knowledge Base 63

6.1. Putting the Knowledge Base On-line 63
6.2. The EMYCIN Executive 64
6.3. The Knowledge Base Editors 65
6.4. Saving the Knowledge Base 66
6.5. Listing the Knowledge Base 71

7. Debugging the Knowledge Base 75

7.1. Consultation Tracing 75
7.2. WHY and HOW Questions 75
7.3. The EXPLAIN, TEST, and REVIEW Options 77
7.4. The QA Module 81
7.5. Debugging with Batch 84

8. Consultation Options 89

8.1. Consultation Special Options 89
8.2. Maintaining Consultant Cases 92
8.3. Controlling the Interaction of the Consultant 93

9. Implementation Details 101

9.1. Miscellaneous System Variables 101
9.2. Pointers to and from Rules 101

9.3. Internal Data Structures during a Consultation 102
9.4. Modifying QA for your domain 104
9.5. The Dictionary 105
9.6. Defining New Predicate and Action Functions 107

Appendix I. Functions 111

I.1. Non-Numeric Predicate Functions 111
I.2. Numeric Predicate Functions 114
I.3. Conclusion Functions 116
I.4. Auxiliary functions 118
I.5. Functions for TEXT-valued Parameters 119
I.6. Mapping functions 121

Appendix II. An Introduction to the Interlisp editor 127

Index 135

List of Figures

Figure 1-1: The major roles of EMYCIN 4
Figure 1-2: A sample rule from the domain of structural analysis 6
Figure 1-3: Example of ARL format for rule input/output. 10
Figure 2-1: SACON's static tree of context types 16
Figure 2-2: An instance tree from SACON 16
Figure 2-3: MYCIN's static tree 18
Figure 2-4: A tree of instances during a particular MYCIN consultation 19
Figure 2-5: LITHO's static tree and an instance tree 19
Figure 2-6: CLOT's static tree and an instance tree 20
Figure 2-7: GRAVIDA's static tree and instance tree 20
Figure 2-8: An example of a dynamic relationship between context instances 21
Figure 2-9: Prompt properties required by a context 24
Figure 3-1: Parameter Editor commands 30
Figure 4-1: A sample rule from the CLOT system. 39
Figure 4-2: CFCOMBINE 44
Figure 4-3: MYCIN's static tree 45
Figure 4-4: ARL Operators 53
Figure 4-5: ARL Forms 53
Figure 5-1: SACON Inference Structure 56
Figure 5-2: A Goal Rule for SACON 57
Figure 6-1: EMYCIN Executive commands 64
Figure 6-2: Property Editor commands 66
Figure 6-3: Migration of Changes 68
Figure 6-4: Knowledge base files for a consultant 70
Figure 7-1: QA Options 82
Figure 7-2: Legal question forms for QA 83
Figure 8-1: Special Options Available Before a Consultation 89
Figure 8-2: Instructions for Running Consultations 90
Figure 8-3: Possible Answers During a Consultation 91
Figure 8-4: Special Options during TAB 95
Figure 9-1: Non-Numeric Predicate Functions 112
Figure 9-2: Interlisp Editor Commands 131

Preface

This document describes the basic knowledge structures and operation of the EMYCIN system. It has been oriented toward the prospective system builder and contains substantial details about the available knowledge acquisition aids.

The first chapter contains a brief introduction to the EMYCIN system and includes descriptions of some recently incorporated features such as rule checking and the Abbreviated Rule Language. We recommend that you read this first chapter before exploring the rest of the manual, even if you are familiar with the EMYCIN system. In addition to refreshing your memory, it serves as a map to the rest of the manual's contents.

The basic building blocks of a knowledge base (contexts, parameters, and rules) are described chapters 2, 3, and 4 respectively. You should be familiar with each of these components before trying to create a knowledge base. The predicate and action functions of rules are described in appendix I.

Much of the material discussed in these chapters is highly interconnected, and certain amount of forward referencing cannot be avoided. Thus, some sections will be difficult to appreciate until later ones have been read. A second reading of these sections will be to one's advantage.

We assume that the reader has a modest familiarity with the Lisp language and, in particular, with the Interlisp dialect [Teitelman 78]. Most data structures in the system are entered in a Lisp format, although they are not necessarily maintained in that exact form internally. Some ability to use the Interlisp editor is very useful, as that is the principal means for editing Lisp structures. We have provided a brief introduction to the structure editor (see appendix II). Further knowledge of Interlisp is useful, especially the break package and the file package, but not strictly essential. For an introduction to the Lisp language see [Charniak 80] or [Horn 81].

Text followed a "←" is to be typed to the Interlisp executive (whose prompt character is ←). This character prints as a "_" on some terminals. The prompt for the Interlisp structure editor is "*". Words in lower case and surrounded by brackets are symbols that are meant to be replaced by an appropriate value. "<cr>" denotes a carriage return; "↑x" denotes control-x for any letter x, e.g. ↑G rings the bell at the terminal.

Example output from a terminal session is printed `in this font`. Text within these examples displayed in **bold face** represents input typed by the expert or the client. Comments on the examples are displayed in *italics*.

Acknowledgments

Many people have contributed to the construction and development of the EMYCIN system, through code and ideas. They include Edward H. Shortliffe, Bruce G. Buchanan, Lawrence M. Fagan, Janice S. Aikins, William J. Clancey, Randall Davis, Miriam B. Bischoff, Alain Bonnet, Sharon Wraith-Bennett, and the entire SUMEX staff.

Several people read and made comments on earlier drafts of this manual: Alain Bonnet, Bruce G. Buchanan, Val Catanzarite, David Goldman, Jacques Harry, Clifford R. Hollander, and Steven Snyder.

The authors would like to especially thank Thomas Dietterich for giving a final draft of this manual a particularly thorough reading and for suggesting (and drafting) countless important improvements.

1. Introduction

Much of the current research in artificial intelligence focuses on computer programs that aid scientists with complex reasoning tasks. Recent work has indicated that one key to the creation of intelligent systems is the incorporation of large amounts of task-specific knowledge. Building such "knowledge-based" or "expert" systems from scratch can be very time-consuming, however, Thus research has also concentrated on the design of general tools to aid the construction of knowledge-based systems.

This manual describes a domain-independent framework for constructing one class of expert programs: rule-based consultants. The system, called EMYCIN,[1] is based on the domain-independent core of the MYCIN program [Shortliffe 76].

The Task

EMYCIN is used to construct and run a *consultation program*, a program that offers advice on problems within its domain of expertise. The consultation program elicits information about a particular problem (a "case") by asking questions of a user. It then applies its knowledge to the specific facts of the case and informs the user of its conclusions. The user is free to ask the program questions about its reasoning in order to better understand or validate the advice given.

For example, the MYCIN system was constructed to advise a physician on the selection of an appropriate anti-microbial therapy for a patient with an infection. In order to suggest any drugs MYCIN reasons about the type of the patient's infection and about the organisms that might be causing it. MYCIN bases its reasoning on data the physician supplies regarding laboratory tests, prior medical history, and any current or prior therapy the patient has received.

In a non-medical application, the SACON consultant was constructed to advise a structural engineer on the use of a very complex computer program, MARC, which uses a finite-element method to simulate a mechanical structure's response to various load conditions. SACON recommends different packages and options available in the MARC system based on an estimation of the kinds of stress and deflection behaviors that are likely to arise for a structure. This estimation is based on, in turn, a description of the structure itself, its geometry and construction, and a description of the various loads (such as braking, bending, etc.) that it will sustain.

There are really two "users" of EMYCIN, as depicted in Figure 1-1. The *system designer* or *expert* interacts with EMYCIN to produce a *knowledge base* for the domain. The knowledge base is made up of two basic components: factual knowledge about the case under consideration and rule-based knowledge about how to carry out the consultation. EMYCIN interprets this knowledge base to provide advice to the *client* or *consultation user*. Thus the combination of EMYCIN and a specific

---

[1] "Essential MYCIN", i.e., MYCIN stripped of its domain knowledge. EMYCIN is written in Interlisp , a programming environment for a particular dialect of Lisp language, and runs on a DEC PDP-10 or -20 under the TENEX or TOPS20 operating systems. The current implementation of EMYCIN uses about 45K words of resident memory and an additional 80K of swapped code space. The version of INTERLISP in which it is embedded occupies about 130K of resident memory, leaving approximately 80K free for the domain knowledge base and the dynamic data structures built up during a consultation.

knowledge base of domain expertise is a new consultation program.

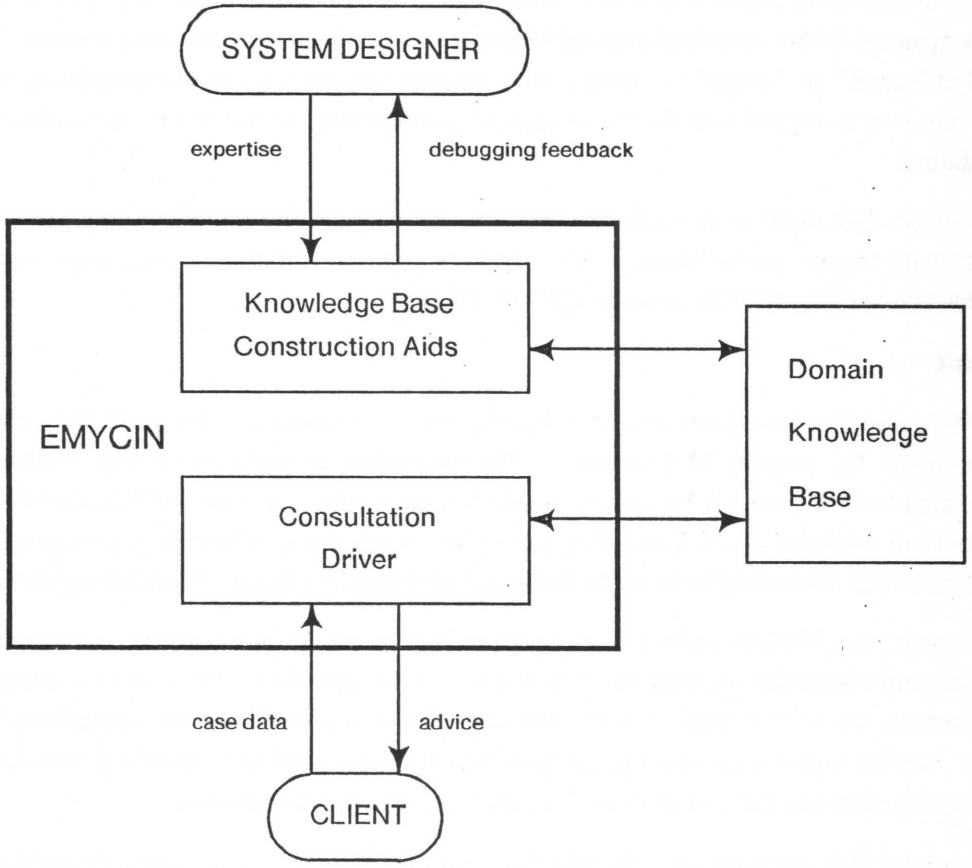

Figure 1-1: The major roles of EMYCIN

Knowledge Organization

Facts

The factual knowledge about a case is stored much as information is stated in traditional databases, as context-parameter-value triples. A *context* is an actual or conceptual entity in the domain of the consultant, e.g., a patient, an aircraft, or an oil-well. Associated with each context is a set of *parameters* such as the age, sex, and race of the patient or the location, depth, and estimated lifetime of the oil-well. Each parameter of each context can take on *values*; the RACE parameter of the patient context could take on values such as caucasian, black, asian, etc.

In terms of traditional programming languages, a context is like a record structure and the parameters are the variables that make up that record structure. The main goal of a consultation, then, is to fill in the values for a few key parameters, such as the organisms causing the patient's

infection or stress behaviors to be observed in the structure. To accomplish this, however, it is often necessary to fill in values for other parameters and so on.

The expert uses the EMYCIN knowledge acquisition aids to define these record structures (contexts, parameters, and values) and then specifies in the production rules how to create instances of these record structures during a consultation and how to fill in the values of the record structures variables.

An important difference between traditional record structures and EMYCIN's factual knowledge structure is that parameters can take on multiple values. The uncertainty of data or competing hypotheses is represented by a number called certainity factor (CF) attached to each fact triple. The CF is a number between -1 and 1 indicating the strength of the belief in, or the importance of, that fact. A CF of 1 represents total certainty, while a CF of -1 represents certainty in the negation of the fact. Although certainty factors are not conditional probabilities, they are informally based on probability theory.[2]

Some sample triples from the SACON system might be

	parameter	context	value	(CF)
(a)	COMPOSITION	*of* SUB-STRUCTURE-2	*is* CONCRETE	(1.0)
(b)	GEOMETRY	*of* SUB-STRUCTURE-3	*is* PLANAR	(-1.0)
(c)	TIME-DEPENDENT	*of* STRUCTURE-24	*is* YES	(1.0)

in other words, (a) the composition of the second sub-structure is concrete, (b) the geometry of the third sub-structure is *not* planar, and (c) there are time-dependent terms in the equilibrium equations of Structure-24. Multiple values for a single context and parameter are permitted if the CF's are less than unity; e.g., alternative hypotheses for the identity of an organism in MYCIN might be

(d)	IDENTITY	*of* ORGANISM-1	*is* PSEUDOMONAS	(.8)
(e)	IDENTITY	*of* ORGANISM-1	*is* E.COLI	(.15),

indicating (d) the identity of ORGANISM-1 is probably Pseudomonas, or, alternatively, that Pseudomonas is important enough that it must be considered further, but (e) there is some evidence to believe it is E.coli.

A useful way of thinking about CF's is to consider them as single numbers combining subjective probabilities and utilities. As such they represent the *importance* of the fact. Either it is highly probable or its consequences carry high costs or benefits. In either case, the fact is important enough to warrant further consideration.

---

[2] At first glance, the concept of a certainty measure associated with a belief or an inferential statement appears to be inherently probabilistic and, accordingly, amenable to analytical techniques such as Bayes' Theorem. The serious limitations of a model based on conditional probabilities, however, have been discussed in detail in the paper that describes the CF model of inexact reasoning [Shortliffe 75].

Rules

The system reasons about a domain using knowledge encoded as *production rules* :

if *premise* **then** *action* (CF).

Each rule has a *premise*, which is a conjunction of predicates over the fact triples in the knowledge base. Rules also have a CF, reflecting the degree to which the premise facts imply the action facts. If the premise is true, the conclusion in the *action* part of the rule is drawn, updating one or more fact triples in the knowledge base. The strength of the conclusion is modified using the CF's of the fact triples in the premise and the CF of the rule.

Figure 1-2 shows a typical rule from the domain of structural analysis (SACON). The English version of the rule is shown first; this is the form in which the user of the consultation program sees the rule. The second version is the internal Lisp form manipulated by the program; the English version can be generated from the Lisp form on demand. The predicates, e.g., LESSP*, GREATERP*, SAME, are simple Lisp functions operating on fact triples and returning CF's as results. Conjunction of the facts in the premise clauses is accomplished by the Lisp function $AND, the multivalued analogue of the Boolean AND function. It performs a minimization operation on CF's, and passes the resulting certainty measure to the action portion of the rule.

RULE068

```
If:  1) The material composing the sub-structure is one of
        the metals,
     2) The analysis error (in percent) that is tolerable is
        less than 5,
     3) The non-dimensional stress of the sub-structure is
        greater than .5, and
     4) The number of cycles the loading is to be applied is
        greater than 10000
Then: Fatigue is one of the stress behavior phenomena in the
           sub-structure.

PREMISE:  ($AND (SAME CNTXT COMPOSITION (LISTOF METALS))
                (LESSP* (VAL1 CNTXT ERROR)
                        5)
                (GREATERP* (VAL1 CNTXT ND-STRESS)
                           .5)
                (GREATERP* (VAL1 CNTXT CYCLES)
                           10000))
ACTION:   (CONCLUDE CNTXT SS-STRESS FATIGUE TALLY 1000)
```

Figure 1-2: A sample rule from the domain of structural analysis

At any given time during a consultation, EMYCIN is working toward the goal of establishing the value of some parameter of a context; this operation is termed *tracing* the parameter. To this end, the system retrieves the (precomputed) list of rules whose conclusions bear on the goal. SACON's RULE068 (Figure 1-2), for example, would be one of the rules retrieved in the attempt to determine the stress of a substructure. Then for each rule in the list, EMYCIN evaluates the premise; if true, it makes

the conclusion indicated in the action.

In the course of evaluating the premise of one of these rules, some clause might not yet be known, i.e., the context/parameter pair tested in the clause may not yet have been traced. In this case, EMYCIN suspends interpretation of the rule while it satisfies the subgoal of finding out the unknown parameter; this may in turn cause other rules to be invoked. Thus, the rules are said to "chain backward" from the top-level goal to the low-level data; it is the attempt to achieve each goal that drives the consultation. The whole process begins by tracing the top-level "goal" parameter, generally the desired "result" of a consultation (e.g., a diagnosis or recommendation).

The order of the rules in the list is assumed to be arbitrary, and all the rules are applied unless one of them succeeds and concludes the value of the parameter with certainty (in which case the remaining rules are superfluous). The order of application of rules is determined by (a) the logical structure of the chains of rules and (b) the tree of objects described in the context tree. See the section 4.2 for the exceptions to this description of rule interpretation.

Questions are asked during the consultation when the rules fail to deduce the necessary information (or there are no rules at all to conclude it). If the client cannot supply the requested information, it remains unknown, and rules that require it will fail.

This control structure tends to ask only "relevant" questions. Questions are asked only if the information (the value of the parameter) is needed by some rule under consideration, and the rules are tried only because they are relevant to some goal the system is trying to achieve. Thus, the questions asked, and the order in which they are asked, may vary from case to case; this contrasts with other kinds of consultation programs that have a pre-specified set of questions that are always asked for all cases.

This control structure permits the graceful handling of incomplete information. If the user is unable to supply some piece of data, the rules that need the data will fail and make no conclusion. The system will thus make conclusions, if possible, based on less information. Similarly, if the system has inadequate rules (or none at all) for concluding some parameter, it may ask the user for the value. When too many items of information are missing, of course, the system will be unable to offer sound advice.

There are many advantages to having rules as the primary representation of knowledge. Since each rule is intended to be a single "chunk" of information, the knowledge base is inherently modular, making it relatively easy to update. Individual rules can be added, deleted or modified without drastically affecting the overall performance of the system. The rules are also a convenient unit for explanation purposes, as a single step in the reasoning process can be meaningfully explained by citing the English translation of the rule used.

Applications

Several consultation systems have been written using EMYCIN. The original MYCIN program and the SACON system have already been described. MYCIN is now implemented in EMYCIN, but its knowledge base was largely constructed before EMYCIN was developed as a separate system.

Results of formal evaluations of MYCIN's competence in the domains of bacteremia (bacterial infections in the blood) [Yu 79] and meningitis [Yu2 79] indicate that MYCIN's performance in these areas has begun to approach that of medical specialists.

The PUFF program [Kunz 78] performs interpretation of measurements from a pulmonary function laboratory. PUFF's knowledge base was constructed using an early version of EMYCIN; the system designers were Stanford computer scientists who had previous experience with MYCIN, collaborating with a pulmonary physiologist and biomedical engineers. The data from over 100 cases were used to create some 60 rules diagnosing the presence of pulmonary disease. The rules are used to create a complete report including the input measurements, other patient data, and the measurement interpretation.

The PUFF rule set was expanded and converted into a Basic program to run on a PDP-11 at Pacific Medical Center in San Francisco. In this form, PUFF is the only EMYCIN system being used routinely for other than experimental purposes. Its reports are reviewed by a staff physician before being entered into a patient's record; most reports (95%) are accepted without change.

The HEADMED program [Heiser 78] is an application of EMYCIN to clinical psychopharmacology. The system diagnoses a range of psychiatric disorders and recommends drug treatment when indicated.

CLOT [Bennett 80] is an experimental system dealing with the diagnosis of disorders of the blood coagulation system. Such diagnoses, once the system was completed, could be used by a physician to estimate the severity and cause of a particular episode of bleeding, to evaluate the effects of various anti-coagulation therapies on a patient, and to estimate the pre-operative risk of a patient having serious bleeding problems during surgery.

Several additional consultants have been constructed using EMYCIN, notably GRAVIDA, a consultant to assist a physician during a patient's pregnancy, LITHO, a consultant that interprets the geology surrounding an oil well, and DART, a system that gives advice on computer telecommunication system failures [Bennett 81].

Range of Applicability

EMYCIN is designed to provide consultative advice. The system takes as input a body of measurements or other information pertinent to a case, and produces as output some form of recommendation or analysis of the case. The framework seems well suited for some deductive problems, notably some classes of fault diagnosis, where a large body of input measurements (symptoms, laboratory tests) is available and the solution space of possible diagnoses can be enumerated. It is less well suited for "formation" problems, where the task is to piece together existing structures according to specified constraints to generate a solution.

EMYCIN was not designed to be a general-purpose representation language. It is thus wholly unsuited for some problems. The limitations derive largely from the fact that EMYCIN has chosen one basic, readily understood representation for the knowledge in a domain: production rules applied by a backward-chaining control structure, with facts about the case represented as associative triples.

While the rule representation is quite general, this choice of control structure is unsuitable for problems of constraint satisfaction, or those requiring iterative techniques.[3] Among other classes of problems that EMYCIN does not attempt to handle are simulation tasks, and tasks involving planning with stepwise refinement.

Facilities and Features of EMYCIN

The remainder of this chapter describes several of the important features that the EMYCIN system provides. EMYCIN is best viewed as a collection of several systems, each of which deals with different aspects of the construction, maintenance, and use of a knowledge base. There are editors for each type of knowledge base component, numerous debugging aids, and facilities for creating and using case libraries for a domain. All of these features are available through a convenient executive (see section 6.2).

Explanation Capabilities

EMYCIN's *explanation program* allows the user of a consultation program to interrogate the system's knowledge, either to find out about inferences made (or not made) during a particular consultation or to examine the static knowledge base in general, independent of any specific consultation.

During the consultation, EMYCIN can offer explanations of the current, past, and likely future lines of reasoning. If the motivation for any of EMYCIN's questions is unclear, the client may temporarily put off answering and instead inquire why the information is needed. Since each question is asked in an attempt to evaluate some rule, a first approximation at an answer is simply to display the rule currently under consideration. The program can also explain what reasoning led to the current point, and what use might later be made of the information being requested. This is made possible by examining records left by the rule interpreter, and by inspecting the rules in the knowledge base to determine which are relevant. This form of explanation requires no sophisticated language understanding by the program; it is invoked by simple commands from the client ("WHY" and "HOW" as described in [Davis 76] and section 7.2).

Another form of explanation is available through the Question Answering (QA) Module, which is automatically invoked after the consultation has ended, and which can also be entered during the consultation to answer questions other than those handled by the specialized WHY/HOW commands described above. The QA Module accepts simple English-language questions dealing with any conclusion drawn during the consultation and about the domain in general. Explanations are again based on the rules; they should be comprehensible to anyone familiar with the domain, even if not familiar with the intricacies of the EMYCIN system. The questions are parsed by pattern matching and keyword lookup, using a dictionary that defines the vocabulary of the domain. EMYCIN automatically constructs the dictionary from the English phrases used in defining the contexts and parameters of the domain; the system designer may refine this preliminary dictionary to add synonyms or better tune

---

[3] With a different control mechanism, production rules are quite capable of iterative use. The VM program [Fagan 79], for example, deals with the management of patients receiving assisted ventilation after cardiac surgery.

QA's parsing. The QA module is described in much more detail in [Scott 77] and section 7.4.

Knowledge Acquisition

The system designer's principal task is entering and debugging the production rules and the fact triples in the knowledge base. EMYCIN provides a terse stylized, but easily understood, language for writing rules and several high-level knowledge base editors for each of the additional knowledge structures in the system. These editors perform extensive checks to catch common input errors, such as misspellings, and handle all necessary bookkeeping chores. These facilities allow the system builder to quickly try out new ideas and thereby assess the feasibility of any particular formulation of the domain knowledge into rules.

The Abbreviated Rule Language (ARL; see section 4.5) constitutes an intermediate form between English and pure Lisp. ARL is a simplified Algol-like language that uses the names of the parameters and their values as operands; the operators correspond to EMYCIN predicate and action functions. For example, SACON's Rule 68 (Figure 1-2) could have been entered or printed as:

```
If    Composition = (LISTOF METALS) and
      Error < 5 and
      Nd-stress > .5 and
      Cycles > 10000
Then  Ss-stress = fatigue
```

Figure 1-3: Example of ARL format for rule input/output.

ARL is a shorthand form derived from *ad hoc* notations that we have seen several of our domain experts use to sketch out sets of rules. The parameter names are simply the labels that the expert uses in defining the parameters of the domain. Because of its conciseness, entering rules in ARL is much easier than entering them in either English or Lisp, an important consideration when entering a large number of rules.

As each rule is entered or edited, it is checked for syntactic validity to catch common input errors. By "syntactic" we mean issues of rule form—that terms are spelled correctly, values are legal for the parameters with which they are associated, etc.—rather than the actual information (semantic) content (i.e., whether the rule "makes sense"). Performing the syntactic checks at acquisition time reduces the likelihood that the consultation program will fail due to "obvious" errors. This permits the expert to concentrate on debugging logical errors and omissions.

The purely syntactic checks are made by comparing each rule clause with an internal "function template" corresponding to the predicate or action function used in the clause. Using this template, EMYCIN determines whether the arguments to these functions are correctly filled. For example, each argument requiring a parameter must be filled by a valid parameter of some context, and any argument requiring a value must be a legal value for that parameter. If an unknown parameter is found, the checker tries to correct it with the Interlisp spelling corrector, using a spelling list of all parameters in the system. If that fails, it asks if this is a new (previously unmentioned) parameter. If so, it defines the new parameter and, in a brief diversion, prompts the system builder to describe it. Similar action is also taken if an illegal value for a parameter is found.

A limited semantic check is also performed: Each new or changed rule is compared with any existing rules that conclude about the same parameter to make sure it does not directly contradict or subsume any of them. A contradiction occurs when two rules with the same set of premise clauses make conflicting conclusions (contradictory values or CF's for the same parameter); subsumption occurs when one rule's premise is a subset of the other's, so that the first rule succeeds whenever the second one does (i.e., the second rule is more specific), and both conclude about the same values. In either case, the interaction is reported to the expert, who may then examine or edit any of the conflicting rules.

Information characterizing the parameters and contexts of the domain is stored as properties (on the property list) of the context or parameter being described (see chapters 2 and 3). When a new entity is defined, the acquisition routines automatically prompt for the properties that are always needed (e.g., EXPECT, the list of values expected for this parameter); the designer may also enter optional properties (those needed to support special EMYCIN features). The properties are all checked for validity, in a fashion similar to that of the rule checker.

The Debugging Process

There is more to building a knowledge base than just entering rules and affiliated data structures. Any errors or omissions in the initial knowledge base must be corrected in the debugging process. In EMYCIN the principal method of debugging is to run sample consultations; i.e., the expert plays the role of the client seeking advice from the system and checks that the correct conclusions are made. As the expert discovers errors, the knowledge acquisition facilities described above come into use, allowing the expert to modify existing rules or add new ones.

While the explanation program was designed to allow the consultation user to view the program's reasoning, it is also helpful as a high-level debugging aid for the system designer. Without having to resort to Lisp-level manipulations, it is possible to examine any inferences that were made, find out why others failed, and thereby locate errors or omissions in the knowledge base. The Teiresias program developed the WHY/HOW capability used in EMYCIN for this task [Davis 77][4] .

EMYCIN provides a debugger based on another portion of the Teiresias program. It actively guides the expert through the program's reasoning chain and locates faulty (or missing) rules. The debugger starts with a conclusion made during a test consultation that the expert has indicated is incorrect, and follows the inference chain back to locate the error. More detail on this REVIEW option can be found in section 7.3.

The expert can also inspect the operation of the rule interpreter as it develops a consultation. This tracing facility (see section 7.1) displays assorted information about what the interpreter is doing: which rules it tries, which succeed (and what conclusions they make), which fail (and for what reason), etc. If the expert observes from the trace that a rule succeeded that should have failed, or vice versa, the consultation can be immediately interrupted, rather than waiting till the end and using

---

[4]The principal focus of the Teiresias program, however, was interactive knowledge acquisition. For more details, see [Davis 76].

REVIEW to explore the inference chain.

In any case, once the problem is corrected, the expert can then restart and try again, with the consultation automatically replayed using the new or modified rules.

Case Library

EMYCIN has facilities for maintaining a library of sample cases. These can be used for testing a complete system or for debugging a growing one. The answers given by the consultation user to all the questions asked during the consultation are simply stored away, indexed by their context and parameter. When a library case is rerun, the answer to each question is looked up and automatically supplied; any new questions resulting from changes in the rule base are asked in the normal fashion. This makes it easy to check the performance of a new set of rules on a "standard" case. It is especially useful during an intensive debugging session, since the expert can make changes to the knowledge base and, with a minimum of extra typing, test those changes—effectively reducing the "turnaround time" between modifying a rule and receiving consultation feedback.

The Batch Facility

A problem common to most large systems is that new knowledge entered to combat one set of problems often introduces new bugs affecting cases that once ran successfully. To simplify the task of keeping the knowledge base consistent with cases that are known to be correctly solved, EMYCIN's batch facility (section 7.5) permits the system designer to run any or all cases in the library in background mode. Batch reports whether any changes occurred in the results of the consultation, and invokes the QA Module to explain why they occurred. The system builder must indicate which parameters represent the results or the most important intermediate steps by which the "correctness" of the consultation may be judged. The use of the Batch program can be viewed as a form of additional semantic checking to supplement the checking routinely performed at the time of rule acquisition.

Human Engineering

As has been described, EMYCIN's existing human-engineering features relieve the system builder of many of the tedious concerns of acquiring and debugging a knowledge base. It is also important of course, that the resulting consultation program be pleasing in appearance to its user. Since the main mode of interaction between the consultation program and the client is in the program's questions and explanations, a number of features concentrate on making that interface as comfortable as possible. The main feature in this category that has been described so far is the explanation program—the client can readily find out why a question is being asked or how the program arrived at its conclusions.

In most places, EMYCIN allows the client to type a question mark to obtain help concerning the options available when prompted for input. When the program asks for the value of a parameter, EMYCIN can provide simple help by listing the legal answers to the question.

The designer may control the manner in which questions are phrased and rephased. By specifying the PROMPT for each parameter that is asked, more detail can be specified than would appear in a

prompt generated by the system from the parameter's translation. Additional help can be given by rephrasings or elaborations on the original question when it is not understood by the client (the REPROMPT property of the parameter, see section 3.1). This capability allows simple questions to be given for experienced clients but provides more detailed explanations for those who are new to the consultation program.

EMYCIN supplies a uniform input facility that permits the normal input editing functions—character, word, and line deletions. It performs spelling correction and Tenex-style completion (after Altmode) from a list of possible answers; typically this is the list of legal values for the parameter being asked about, as supplied by the system designer.

2. Contexts

As we noted in the introduction, an EMYCIN knowledge base is composed of factual knowledge about the domain and the case and production rules that control the consultation interaction and make inferences about a case. Factual knowledge is represented in the form of fact triples, (context parameter value), that are modified by certainty factors.

The context types and their parameters selected by the expert for the consultant are an important part of the expertise of the domain. They reflect the *structural* aspects of the expert's problem-solving viewpoint. These constructs form the vocabulary used to express both the rules of inference and the expert's problem-solving methods for the domain. This chapter describes the internal structure of the contexts and introduces the idea of the context tree; later chapters will describe parameters, their values, and their use in production rules.

2.1. The Context Tree

Of all the structures the expert must specify for an EMYCIN consultant, the context tree is perhaps the most important, yet least discussed, component of the knowledge base. The context tree forms the backbone of the consultant, organizing both the conceptual structure of the knowledge base and the basic flow of the consultation interaction. The tree also indicates the goals that the consultant will initially attempt to determine values for. Since the principles for designing new context trees are poorly understood, this chapter is populated with examples from various existing EMYCIN systems. It is hoped that these examples will help system builders make appropriate choices for their own domain-specific contexts.

The context tree is composed of at least one, but possibly many, context types. A context type corresponds to an actual or conceptual entity in the domain of the consultant, e.g., a patient, an aircraft, or an oil-well. Each context type in the context tree is very much like a record declaration in a traditional programming language. It describes the form of all of its instances created during a case. Thus, there are two related but distinct aspects of the context tree mechanism: a static tree of *context types* and a dynamic tree of *context instances*. The static tree of types is the structure defined by the expert during system construction and forms the knowledge base "core." This chapter describes this structure and offers hints on its design.

The static tree is used to guide the creation of the dynamic context tree of instances during the consultation. These instances are also organized into a tree that has a form mimicking the structure of the static hierarchy. In this chapter, and in the rest of the manual, we will distinguish these two structures by referring to them as the static tree and instance trees respectively. A moderately complex example of both these types of trees for the SACON system is given in the figure below. In the figures below, the links, or relationships, among context types are sometimes labelled to show different uses of the tree.

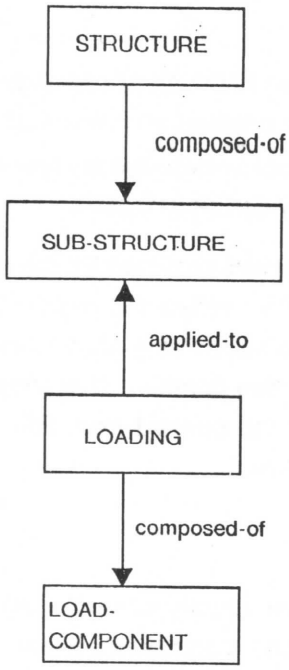

Figure 2-1: SACON's static tree of context types

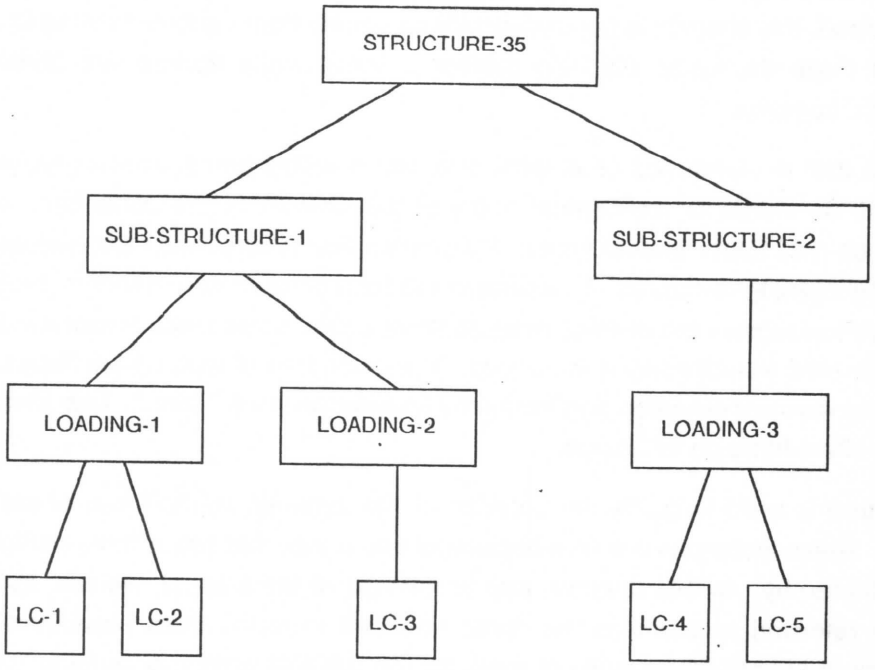

Figure 2-2: An instance tree from SACON

Each knowledge base has at least one context type corresponding typically to the entity about which a consultation gives advice. This is the main, or root, context type of a consultation and there will be a single instance of it for each consultation. In MYCIN, for example, the main context type is PATIENT and a consultation provides advice about disease(s) of the patient. In SACON, the main context type is STRUCTURE and a consultation gives advice about performing structural analysis on a structure (such as a bridge or an airplane wing).

Some domains are simple enough that no other context types are needed. In the PUFF system, for example, only attributes of the patient were needed. However other systems, such as MYCIN and SACON, require the ability to discuss attributes of multiple objects. In these cases, the context types are organized into a simple tree structure with the main context at the root. For each context type that is subordinate to another context type there is an implicit one-to-many relationship between the instances of each type created during a consultation. Thus, for SACON, there can be many SUBSTRUCTURE instances for the single STRUCTURE instance during a case and there can be several LOADING instances for each SUBSTRUCTURE instance. It should be noted that, except for the root type, context types need not necessarily be instantiated during a particular consultation. In the MYCIN system, for example, the patient may or may not have had any prior drug therapy.

The static tree is the major repository of structural and control information about the consultant. It indicates, in particular, the possible parameters of a context (its PARMGROUP) and the groups of rules that can be applied to instances of a context (its RULETYPES) . Hence, the context types must be defined before you can proceed to acquire rules and parameters, since these are defined with respect to the context tree. In addition, the static relationships between the context types dictate, in large part, the basic mechanism for the propagation of the dynamic tree of instances during a consultation (see [Shortliffe 76]).

All of the production rules used by the consultant to reason about the domain are written without regard to specific context instances in an actual consultation. A rule instead refers to parameters of certain context *types*, and the rule is applied to all of the context instances for which its parameter group is relevant. A collection of applicable context types is termed a rulegroup and determines a rule's SUBJECT (see section 4.3).

For example, a rule that concludes about a parameter of a LOADING, say FORCE-BOUND, will be applied to all the instances of LOADING shown above (e.g. LOADING-1, LOADING-2,) and may or may not succeed within each instance depending on whether its premise is true in that particular context. In addition, if a rule refers to a specific context type, its premise can refer to the parameters of any direct ancestors of this context type. Continuing with our example, the rule premise could refer to parameters of both the SUB-STRUCTURE's and the STRUCTURE itself. The instance tree organization makes clear which LOADING instances are associated with which SUB-STRUCTURE instance. Thus, if the rule concluding about FORCE-BOUND in the LOADING-2 instance mentioned the GEOMETRY of a SUB-STRUCTURE, it would use the GEOMETRY of SUB-STRUCTURE-1. When concluding about the FORCE-BOUND in the LOADING-3 instance, the GEOMETRY of SUB-STRUCTURE-3 would be used.

A single rule may refer to parameters of several context instances above it in the tree. If a rule is

applied to some context instance and uses information about context instances lower in the tree, however, an implicit iteration occurs: the rule is applied to each of the lower instances in turn. If the lower context types have not yet been instantiated, they are at this time. Thus, contexts are instantiated because rules need them,[5] just as parameters are traced when rules need them. In fact, since the goals of the consultation usually consist of finding out something about the root of the tree, the only way that lower context types are instantiated at all is through the application of rules that use information about lower context types.

2.2. Uses of the Context Tree

There have been a few rather stereotypic uses of the additional context types in a context tree and we describe them below. While by no means exhausting the possible uses of the context tree, we hope they will aid the expert and knowledge engineer select appropriate context types and their organization.

The primary use of additional contexts has been to ***structure the data or evidence*** required to advise the client about the root context. Thus, in the MYCIN system, the various types of culture contexts describe the diagnostic tests the doctor performed to isolate the organisms of causing the infection in the PATIENT. Additional information about the patient's current and previous therapies, the cultures and MYCIN's own estimation of the suspected-infections are also represented in the tree.[6] The current context organization for this system is given below.

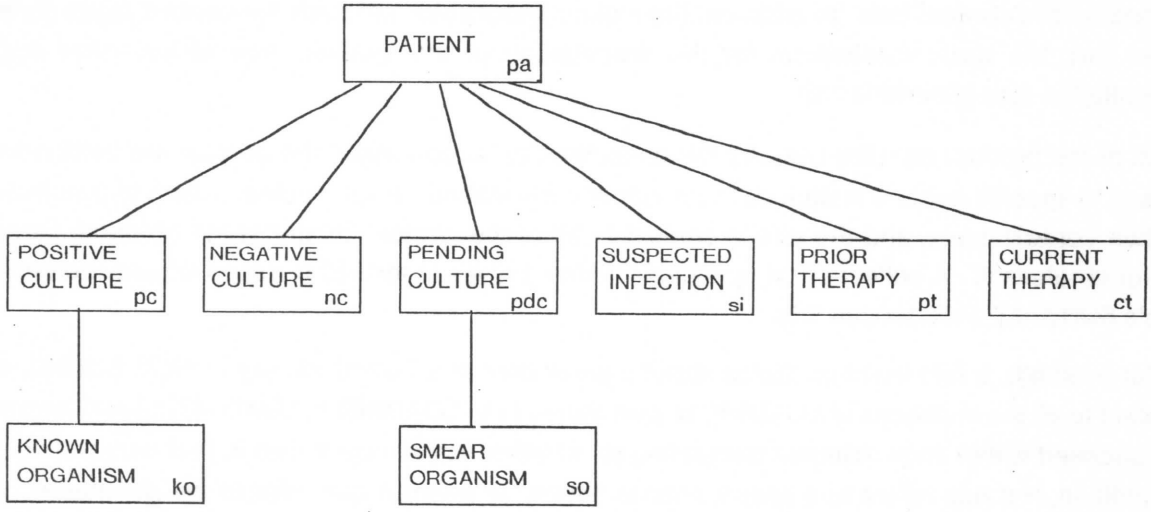

Figure 2-3: MYCIN's static tree

---

[5] Contexts may also be instantiated by explicit command, but the mechanism is less convenient.

[6] It is instructive to compare this structure with the original context tree described in [Shortliffe 76]; the MYCIN system has undergone at least 3 intermediate (and painful!) re-organizations of its static tree. Signficantly however, the *kinds* of objects in the tree have not changed substantially.

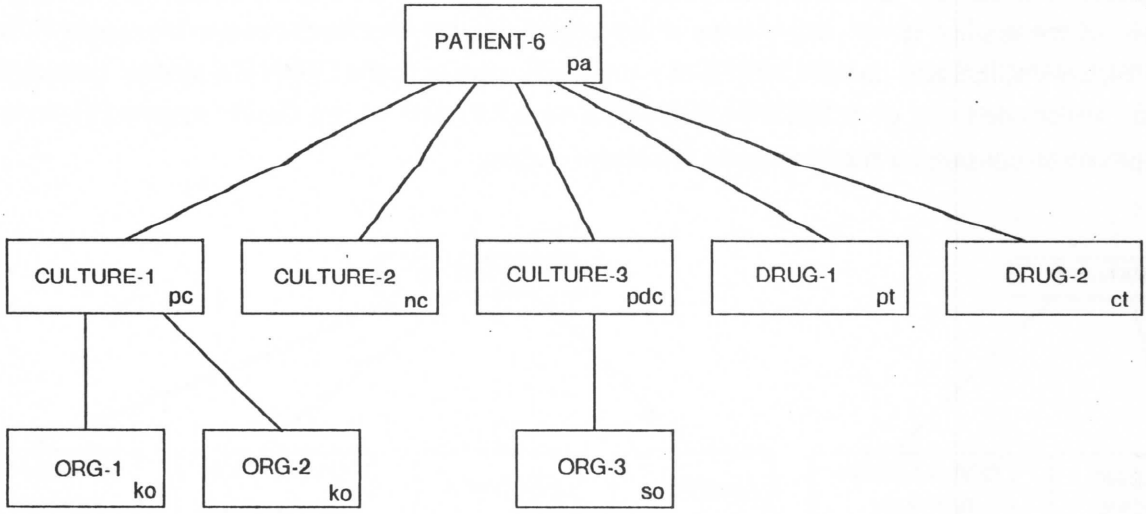

Figure 2-4: A tree of instances during a particular MYCIN consultation

The second major use of the context tree has been to ***organize the distinguished components of some object***. For example, in the SACON system the sub-structures of the main structure correspond to components or regions of the object that have some uniform property, typically a specific geometry or material. Each sub-structure instance is considered *independently*, and conclusions about individual responses to stress loadings are summarized on the structure level to provide a "global" sense of the overall response of the structure. A recent, additional example of this use of a part-whole hierarchy is found in a system called LITHO, which interprets data from oil wells. In this system, each well is decomposed into a number of zones that the petrologist can distinguish by depth [Bonnet, personal communication].

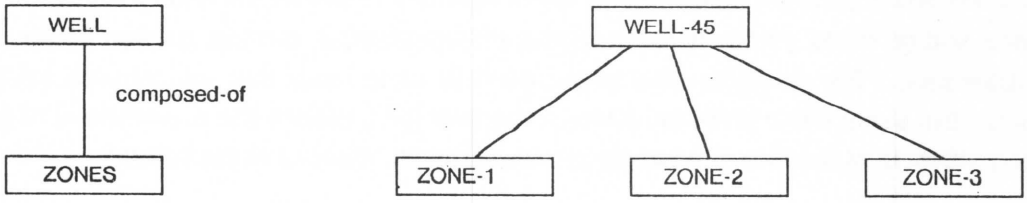

Figure 2-5: LITHO's static tree and an instance tree

A context need not correspond to some physical object but may be an abstract entity. However, the relationships among contexts are explicitly fixed by the arrangement of the context type tree. For this reason, physical objects, represented in this *part-whole* fashion, lend themselves more readily to the current context tree mechanism.

The last major use of the context tree, which is closely related to the part-whole use above, has been to ***represent distinguished events or situations*** that happen to an object. Thus, in the SACON system, a LOADING describes an anticipated scenario or maneuver (such as pounding or braking) to which the particular SUB-STRUCTURE is subjected. Each LOADING, in turn, is

composed of a number of independent LOAD-COMPONENTS, distinguished by the direction and intensity of the applied force. Other uses of this organizational idea have been to represent individual past PREGNANCIES and current VISITS of a pregnant woman in the GRAVIDA system (unpublished) and the anticipated use of BLEEDING-EPISODES of a PATIENT in the CLOT system [Bennett 80].[7] The context structures for these systems are shown below.

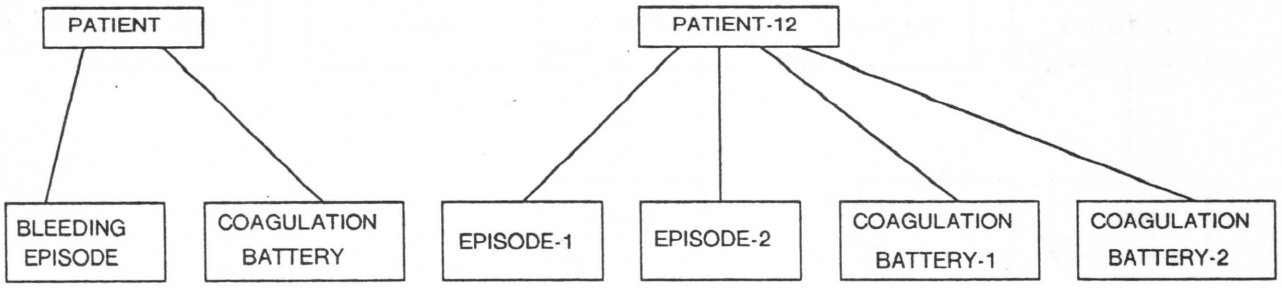

Figure 2-6: CLOT's static tree and an instance tree

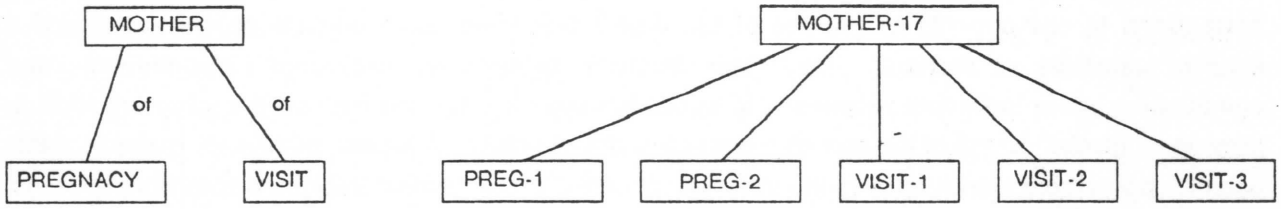

Figure 2-7: GRAVIDA's static tree and instance tree

The primary reason for defining additional context types in a consultant is to represent multiple instances of an entity during a case. Some users may like to define context types that always have one instance and no more, primarily for purposes of organization, but this is often unnecessary (and even cumbersome). For example, one might want to write rules that use various attributes of a patient's liver, but since there is always exactly one liver for a patient there is no need to have a liver context; any attribute of the liver can simply be viewed as an attribute of the patient.

Reference to parameters of contexts in *different* parts of an instance tree, which may be required to make some inferences, is currently very awkward. For example, in MYCIN, a particular drug may be associated somehow with a particular organism (see the figure below). However, this relationship between context instances is *not* one that always holds between all organisms and all drugs: Not all drugs are prescribed to treat all identified organisms. This "prescribed for" relationship cannot be stated statically, independent of the case. Special predicate and action functions must be written to establish and manipulate these kinds of relationships between instances. It is best to avoid these interactions between disjoint parts of the tree during the initial knowledge base design.

---

[7] It should be noted that use of the context mechanism to handle sequential visits in the GRAVIDA system is experimental and required the definition of numerous additional functions for this purpose. They are not currently in EMYCIN.

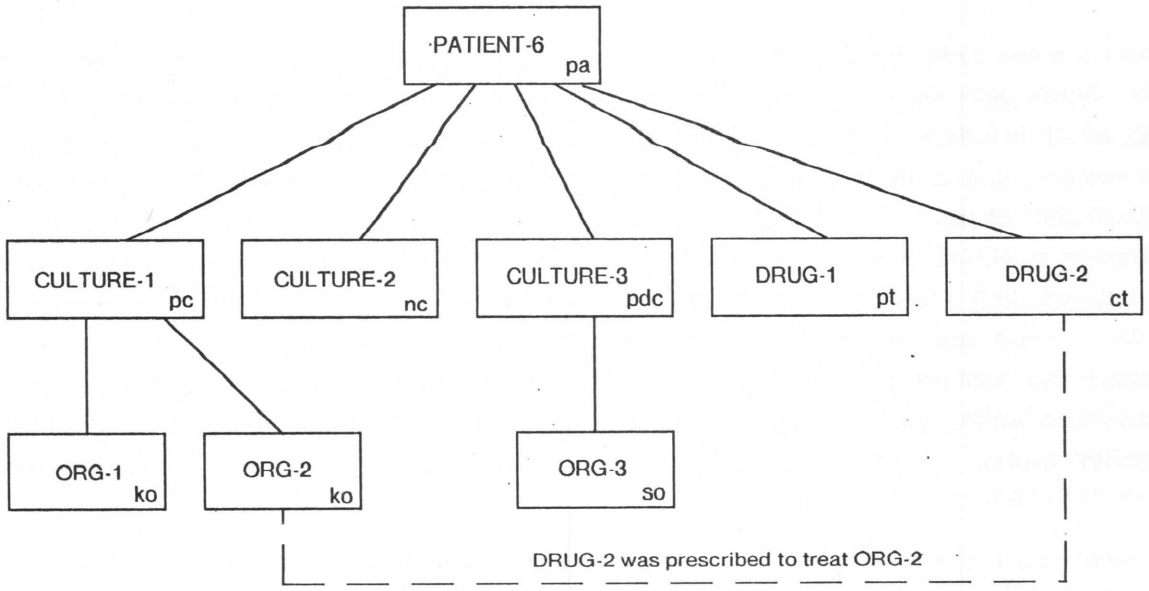

Figure 2-8: An example of a dynamic relationship between context instances

Summing up our experience with this mechanism and considering its relative inflexibility, we offer this final caveat: For an initial system design, start small, and use only one or two context types. Plan the structure of your consultant's context tree carefully before running the EMYCIN system, since restructuring a context tree is perhaps the most difficult and time-consuming knowledge-base construction task. Indeed, restructuring the context tree implies a complete restructuring of the rest of the knowledge base.

2.3. Defining Contexts

When you first enter EMYCIN, you will be asked if you wish to create a new knowledge base. If so, you will automatically be put in the context acquisition routines to define the context types of your domain. You will be prompted for the entire static tree, one context type at a time, in a depth-first fashion.

If you want to add additional context types to an existing static tree, invoke the context editor with the "Contexts" command from the EMYCIN executive (see 6.2). The context editor prompts for a context type. Currently all you can do is define new context types. In any case, when you define a new context type, there are several properties you must specify. For a few of these properties, the system is able to construct a default value from previous responses. You can accept the value offered, edit it, or enter another value if you want to phrase it differently. You must enter the INITIALDATA and GOALS properties from scratch, since EMYCIN has no clue as to what they might be; if they contain new parameters, you will be asked to define them briefly (see section 3 for details of parameter acquisition). Properties of existing context types can be modified using the normal parameter editor.

2.4. Parameter Groups and Rule Groups

There are two additional, important, knowledge-base structures that are related to the definition of context types: *parameter groups* and *rule groups*. Each context type has an associated parameter group, which is indicated by the PARMGROUP property of the context. The parameter group is an atom whose value is the list of parameters that are relevant to the context. For example, the PARMGROUP of the PATIENT context in the MYCIN system is PT-PARMS and it contains such parameters as NAME, AGE, SEX, etc. When applying a rule, the system uses this property to tell which context in the tree a particular parameter belongs to. The PARMGROUP atom (and the value list) are created automatically during the definition of the context type. When defining a new parameter (for example, using the parameter editor) you will be asked to specify its parameter group; this indicates which context types the parameter can apply to. A parameter can belong to only one parameter group. In addition, context types themselves belong to the "parameter group" CONTEXTTYPES.

In most cases each context type in a system will have associated with it a different set of parameters and, hence, each will have a different PARMGROUP. In some cases however your context types will share many characteristics. It is possible for several context types to use the *same* parameter group, as long as the contexts are in different parts of the tree (one is not the ancestor of the other). For example, MYCIN has 4 different *kinds* of cultures (positive cultures, negative cultures, pending cultures, and "non-cultures") that all use the parameter group CULTURE-PARMS.

In addition to defining the parameter group of each context type, you may also define the groups of rules that will apply to a context type and its ancestors in the static tree. If you have defined a context type using the context acquisition routines, a RULEGROUP is automatically defined that will contain all the rules that can apply to this context type and its ancestors. This rulegroup is added to the RULETYPES property of the context type.

If you have context types that share a parameter group, you will probably want to organize your rules into groups that apply to different subsets of the context types that share the parameter group. In MYCIN , for example, the rulegroup CULRULES contains the rules that apply to all four culture context types (and the patient), REALCULRULES contain those applying to the patient and the positive, negative, and pending cultures; PENDCULRULES contain those applying only to the patient and pending cultures. See section 4.3 on how to set up these additional groups.

2.5. Properties of Context Types

This section describes the properties a context type can have. A context must have a TRANS, PRINTID, and PARMGROUP property and typically has an INITIALDATA and GOALS property. The root context, for example, should always have a GOALS property defined—the consultation is driven by tracing the parameters on this list.

The example below shows the properties of the context type POSCUL from the MYCIN system.

```
POSCUL [CONTEXTTYPES]
```

```
TRANS:  (positive cultures obtained from *)
PARMGROUP:  PROP-CUL
PRINTID:  CULTURE-
INITIALDATA:  (SITE WHENCUL COLLECT)
GOALS: (INFECTION)
PROMPT1ST:  (Are there any cultures for * which may be related to the
              present illness, and from which organisms have been
              grown successfully in the microbiology laboratory?)
PROMPT2ND:  (Are there any other cultures from which organisms have
              been successfully isolated in the laboratory?)
ASSOCWITH:  (PATIENT)
OFFSPRING:  (KNOWNORG)
RULETYPES:  (CULRULES POSCULRULES)
SYN:  (((SITE ; WHENCUL)
       (the SITE culture ; of WHENCUL)))
UNIQUE:  MAYBE
CNTXTSUMMARY:  ((SITE WHENCUL)
                (((SITE WHENCUL WHENPOS)
                  (This SITE culture was obtained on WHENCUL and
                   turned positive on WHENPOS .))
                 ((SITE WHENCUL)
                  (This SITE culture was obtained on WHENCUL .)))
                (NIL COLLECT ABDOWOUND INFECTION))
TABPARMS:  (SITE WHENCUL COLLECT)
LEGALTERM:  (2 3)
TABHEAD:  ("site" ("date" "collected")
                  ("method of" "collection [urine,sputum only]"))
TABSTOPS:  (0 15 32 62)
TERSEHEAD:  ("site" "date collected" "coll. method")
```

TRANS — How to translate the context in rules and questions. The TRANS is a list of words; if it contains an asterisk, the latter will be filled at output time with the translation of the parent context, e.g., (the positive cultures of *). There can only be one asterisk. If you enter the TRANS in all uppercase it will be converted to lowercase for you automatically.

PRINTID — An atom ending in "–" used to form the context identifier (by appending a numeral) for contexts of this type; e.g., in MYCIN the PRINTID of POSCUL (positive cultures) is "CULTURE–".

PARMGROUP — An atom of the form "type-PARMS" that names the parameter group for this type of context.

INITIALDATA — A list of parameters, a subset of the ASKFIRST parameters (see section 3) in the context's PARMGROUP, that the consultant should always ask for when an instance of this context type is created. The client will be asked for the value of each of these parameters as soon as a context is created. This often serves to present a more coherent dialog than would appear if each parameter were requested when it was first needed in a rule.

GOALS — A list of parameters, a subset of parameters in the context's PARMGROUP, that the consultant should "trace" when an instance of this context type is created. These are non-ASKFIRST parameters, representing something you will always want to deduce about a new context, (see chapter 3). The parameters on the

GOALS list of the main (or root) context type are referred to as the goals of the system; the values of these parameters represent the advice that the system will give. Contexts other than the root need not (and usually *do not*) have a GOALS property.

The following three "PROMPT" properties are relevant only to context types other than the root. These properties reflect the expected number of instances of a context type per ancestor instance during a consultation. More importantly, they specify the desired interaction with the user when creating the context instances. Their form is a list (much like PROMPT, see section 9.3), in which the asterisk is replaced by the translation of the parent context.

Specific combinations of these PROMPT properties specify the expected number of instances of the context type. The table below summarizes the possible combinations currently allowed in the system. "Required" indicates that the property should have a value, no entry indicates that it shouldn't, and "Optional" indicates that this property is optional. Note that PROMPTEVER and PROMPT1ST are mutually exclusive options.

Number of Instances	PROMPTEVER	PROMPT1ST	PROMPT2ND
Exactly one instance	Optional	Omitted	Omitted
At most one instance	Omitted	Required	Omitted
At least one instance (and possibly many)	Optional	Omitted	Required
Unknown number (and possibly none)	Omitted	Required	Required

Figure 2-9: Prompt properties required by a context

PROMPT1ST — The prompt asking whether contexts of this type exist. If the user answers Yes, then an instance of this context type will be set up, and its INITIALDATA and GOALS will be determined. For example, the PROMPT1ST of CURTHER is

(Is * currently receiving therapy with any antimicrobial agent?)

If there is no PROMPT1ST, then it is assumed that there is *always* at least one instance of the context type, and the PROMPTEVER (if any) is printed.

PROMPTEVER — The "prompt" that will be printed when the first context of this type is created, in lieu of the PROMPT1ST question. This is not strictly a prompt; it requires no answer, and is used to "announce" the new context instance to orient the user to the INITIALDATA questions that will follow. E.g., in MYCIN, there is always at least one KNOWNORG under every POSCUL (by definition a positive culture is one from which organisms grew), so the PROMPTEVER of KNOWNORG is:

(The first organism isolated from * will be referred to as:)

Note: the root context can have a PROMPTEVER property. This provides a method for making general announcements to the user of the consultant, perhaps to refresh his mind about what will be required of him.

PROMPT2ND — The prompt asking whether additional contexts of this type exist (to be used after at least one context of this type has been created). Omission of this property indicates that there is never more than one instance of this type of context under a given parent context. E.g., the PROMPT2ND of KNOWNORG is

(Were any other organisms isolated from * ?)

PROMPT1ST and PROMPT2ND may simply be T, in which case the system generates prompts from the context type's TRANS prop, of the form (Are there any <context translation> of * ?) and (Are there any other <context translation> of * ?).

The following properties are optional and are used to control the translation of context identifiers during a consultation.

SYN — A template used for translating contexts of this type in questions or rules. Often there are attributes of a context whose values describe the context to the user better than the naked identifier (e.g. "ORGANISM-3"). The SYN property indicates how to arrange such attributes into a phrase describing the context.

In its simplest form, the SYN property uses just one parm, and looks like (((parm) (a phrase containing parm))) [i.e. a list of one element, which itself is a list of two elements], e.g. for a culture in MYCIN one might write (((SITE) (the SITE culture))). If the parameter is itself the "name" of the context, then the phrase is "parameter" alone. E.g. in MYCIN the SYN of the patient is (((NAME) (NAME))).

More generally, the SYN property is a list of one or more entries (<parms> <form>), where <parms> is a list of one or more parameters of the context, and <form> is a simple list of words including the elements of <parms>. The parameters must appear in the same order in both <parms> and <forms>, and they must all be ASKFIRST . An asterisk is used in <parms> and <form> to represent the parent context; the atom ";" indicates places where an unambiguous translation can stop (see below). It is intended that if there is more than one entry to the SYN that each successive entry denotes successively "weaker" or less desirable translations, to be used only if the stronger phrases cannot be composed.

To create a context's translation, the system scans the SYN property until it finds an entry for which it knows the values of all parameters in <parms>; values which were not supplied by the user will not be used (since the translation is intended to be "in the user's terms"). Once an element of SYN has been selected, a translation will be constructed by replacing each parameter in <form> with that parameter's value; * will be replaced by the translation (using the SYN property) of the parent context. If a semicolon is encountered and the translation so far is unambiguous, i.e. there are no other contexts with the same value for the slot(s), then the translation process is finished. E.g., the element from the SYN property of an organism ((IDENT ; *) (the IDENT ; from *)) might result in "the Klebsiella" if there is no other organism whose IDENT is Klebsiella, and "the Klebsiella from the blood culture" if there is.

If there is no SYN prop, the context identifier (e.g., ORGANISM-1) will remain untranslated.

UNIQUE — For use in conjunction with the SYN prop: it controls whether the context identifier needs to appear in the translation. For example, a typical non-unique phrase is

"the blood culture (CULTURE-2)", since it is possible to have more than one culture from the same site. If the UNIQUE property is T, the context identifier is omitted (the root context type, for example, automatically receives a UNIQUE property of T). If UNIQUE is MAYBE, it means to omit the identifier if the first try at translating the context is, in fact, unique; e.g., if CULTURE-2 in the example above were the only blood culture in the consultation, then its translation would be simply "the blood culture". Most context types are "non-unique" and do not have a UNIQUE property. The property is not necessary even on unique contexts; it exists simply to reduce excess verbiage where possible.

Context types may have other optional properties. The context type needs a CNTXTSUMMARY property if you want to use the SUMMARY option, see section 8.3. If parameters of the context type are to be gathered in a block using the TAB option for tabular input, it needs the TABPARMS , TABHEAD , TERSEHEAD , TABSTOPS , and LEGALTERM properties as described in section 8.3.

The following three properties are usually computed automatically by the context and parameter editors, so you should not need to modify them yourself. They are listed here for completeness.

ASSOCWITH — A list of ancestor context types, showing this context type's location in the context tree. E.g. the ASSOCWITH of KNOWNORG in MYCIN is (POSCUL PATIENT) and the ASSOCWITH of POSCUL is (PATIENT). This means that a POSCUL context is directly below the patient context, and that a KNOWNORG context is directly below a POSCUL.

OFFSPRING — A list of descendant context types, indicating which types can hang directly below a context of this type in the context tree. This is, roughly speaking, the inverse of ASSOCWITH.

RULETYPES — a list of all rule groups applicable to this context (see section 4.3), e:g., RULETYPES of POSCUL is the list (CULRULES REALCULRULES POSCULRULES). This is the inverse of the CONTEXT property of rule groups.

3. Parameters

Associated with each context type in the system is a set of attributes or *parameters* that represent properties of the context or describe facts about the case in general. These parameters are stored as a list associated with each context type, called its parameter group (see section 2.4). This chapter describes some uses that have been made of parameters and discusses the internal format of individual parameters and their expected values.

Broadly speaking, parameters have been used in two primary ways. Their first use has been ***to represent the goals or advice*** the consultant is to determine. Thus, for the SACON system, the major goals are to determine the stress and deflection behaviors of the structure, to determine the relative magnitudes of the loadings causing these behaviors, and to determine the possible analysis classes required to model this structure using the MARC program. Each of these goals are represented as one or more parameters in the SACON system, some portions of which are shown briefly below.

```
ANALYSIS-CLASS:   a parameter of a STRUCTURE context
 TRANS ("the analysis class of" *)
 LEGALVALS (LINEAR-ANALYSIS NONLINEAR-GEOMETRY-CRACK-GROWTH
        NONLINEAR-GEOMETRY-STRESS-MARGIN NONLINEAR-GEOMETRY-FATIGUE
        BUCKLING BIFURCATION NONLINEAR-GEOMETRY-EXCESSIVE-DEFLECTION
        STIFFNESS-DEGRADATION ...  NONLINEAR-BUCKLING NONLINEAR-BIFURCATION
        GENERAL-INELASTIC GENERAL-NONLINEAR NO-ANALYSIS)
 MULTIVALUED T

STRESS:    a parameter of a STRUCTURE context
 TRANS ("the stress behavior phenomena in" *)
 LEGALVALS (STRESS-COMPARED-WITH-ALLOWABLES YIELDING-COLLAPSE
           CRACKING-POTENTIAL FATIGUE MATERIAL-INSTABILITIES
            STRESS-EXCEEDENCE)
 MULTIVALUED T

DEFLECTION:    a parameter of a STRUCTURE context
 TRANS ("the deflection phenomena in" *)
 LEGALVALS (EXCESSIVE-DEFLECTION FLEXIBILITY-CHANGES
            INCREMENTAL-STRAIN-FAILURE BUCKLING LOAD-PATH-BIFURCATION
            KINEMATIC-COLLAPSE-LOAD)
 MULTIVALUED T

ND-STRESS:   a parameter of a LOADING context
 TRANS ("the non-dimensional stress of" *))
 EXPECT POSNUMB

ND-DEFLECTION:    a parameter of a LOADING context
 TRANS ("the non-dimensional deflection of" *))
 EXPECT POSNUMB
```

Most of these parameters have a list of expected values (the EXPECT property), which the expert enumerated. The ability of the expert to provide these lists is one of the important indicators of whether EMYCIN is suitable for a particular domain. If the expert is unable or unwilling to give such

lists of values for the goal parameters, one should think twice about using EMYCIN. Perhaps the domain is not formalized enough to express the advice in this form, or the very nature of the solution sought by the consultant is one that requires the iterative composition of a solution based on case-specific data. Domains with these characteristics are not handled well in the current EMYCIN framework.

The second major use of parameters has been ***to represent the data or evidence*** from which the advice will be inferred. Once again, taking examples from the SACON system, parameters were used to describe the geometry, construction, and method of support for the sub-structures, and the direction, point of application, and duration of the loadings applied to each sub-structure. Examples of these "data" parameters are shown below. Parameters of this particular type make up the vast majority of all the parameters elicited from the expert when constructing a consultant.

GEOMETRY: *a parameter of a SUB-STRUCTURE context*

```
TRANS ("the geometry of" *)
PROMPT ("Is the overall geometry of" * "curved or planar?")
EXPECT (CURVED PLANAR)
ASKFIRST T
```

COMPOSITION: *a parameter of a SUB-STRUCTURE context*

```
TRANS ("the material composing" *)
PROMPT ("What is the material composing most of" * ?)
EXPECT ((ALLMATERIALS))
ASKFIRST T
```

SUPPORT: *a parameter of a SUB-STRUCTURE context*

```
TRANS ("the configuration of support for" *)
PROMPT ("What is the configuration of support for" * ?)
EXPECT (ONE-SIDE TWO-OPPOSITE-SIDES TWO-ADJACENT-SIDES THREE-SIDES
        FOUR-SIDES)
ASKFIRST T
```

CYCLES: *a parameter of a LOADING context*

```
TRANS ("the number of cycles" * "is to be applied")
PROMPT ("Enter the number of loading cycles to be applied:")
EXPECT POSNUMB
ASKFIRST T
```

SITE: *a parameter of a LOADING context*

```
TRANS ("the site of" *)
PROMPT ("Describe where on the sub-structure" * "is applied:")
EXPECT (NEAR-CENTROID NEAR-SUPPORT NEAR-FREE-EDGE)
ASKFIRST T
```

DIRECTION: *a parameter of a LOADING context*

```
TRANS ("the surface to which" * "acts normal")
PROMPT ("Which surface of the sub-structure does" * "act NORMAL to? (If
         more than one surface, you should consider the loading as two
         or more component loadings)" )
EXPECT (THICKNESS-LENGTH THICKNESS-WIDTH WIDTH-LENGTH)
ASKFIRST T
```

These facts generally have lists of expected values as well and, thus, the warning mentioned above also applies. In addition, these "data values" should have a clear, well-defined meaning for the prospective client of the consultant. After all, these values will be used to answer the questions the consultant asks.

Another convenient classification of the individual parameters, which is independent of their use, is based on the number and type of values a parameter can take on during a consultation. There are two major classes, *single-valued* and *multiple-valued* parameters. A single-valued parameter is used to represent a concept whose expected values are mutually exclusive. That is, you would normally expect only one of the values to be applicable at any one time.[8] For example, in the MYCIN system, a patient is either male or female, either has a fever or hasn't, has a unique age and race, etc. In contrast, there are parameters that may more naturally be multiply-valued during a consultation. Thus, a patient may have had many different kinds of surgery, be allergic to many different drugs, and be infected by numerous organisms. When doing a structural analysis, a structure might exhibit many different loading behaviors and there may be several possible modelling methods that will serve to analyze these behaviors.

The EMYCIN system uses this single/multiple valued distinction when acquiring the definition of a parameter from the system-builder. Whenever a new parameter is entered, EMYCIN asks the system-builder to classify it as either a "Yes/no", "Singlevalued", "Multivalued", or "Ask-all" parameter type. This classification is used to decide which parameter properties are required or appropriate. These parameter types are described under the specific properties below.

If the parameter does expect one or more of a list of values, there are a few pointers we can offer about their representation and use in a knowledge base. If a particular value is more appropriately referred to by a phrase, it is conventional to either abbreviate it (e.g. ACIDFAST is used for "Ziehl-Neelsen acid fast test") or the phrase is constructed directly using hyphens (e.g. STAPHOLOCOCCUS – COAG – POS for "staphylococcus coagulation positive"). Abbreviations can be explained to the client either in the initial PROMPT or using the REPROMPT mechanism (see below). If these phrases are to be asked, the client can use the automatic completion feature supplied with EMYCIN to avoid having to type these hyphenated responses. Thus, there need be no penalty for having a more readable interaction. In addition, this technique also enhances the readability of the rule translations and thus increases the transparency of any explanations the system generates.

In most domains there are natural subsets of the EXPECT and LEGALVALS lists that the expert might want to use, either in rules or other parameter definitions. In SACON, for example, the material of the sub-structure is a list of 20 specific construction materials of different grades. In addition to this list of "all-materials," the expert wanted to refer to them in rule premise clauses by their major classes: aluminums, steels, concretes, etc. These lists of values can be created using the "Variables" executive command and are typically used in rule premises (see the discussion of the LISTOF

---

[8] In the EMYCIN system, this general notion of mutually exclusive values is modified somewhat by the use of certainty factors. See discussion of the EXPECT property and [Shortliffe 76] for details.

function in section I.4) or when certain parameters share expected values. You must define such lists using the Variables command, to insure that the variable will be recorded on the knowledge base.

3.1. Defining Parameters

There are three ways of entering the parameter acquisition routines. When defining a new context, the specification of INITIALDATA and GOALS will lead to the acquisition of any new parameters (see page 58). Additional parameters are also acquired after their first use in a rule or can be defined by directly invoking the parameter editor from the EMYCIN executive (the "Parameter" command, see section 6.2). The editor prompts for a parameter name and then for its properties. You can also modify context types, rule groups, and other "parameter-like" entities in the parameter editor.

When prompted for a parameter you may enter the name of a parameter (existing or new), or you may type <cr> to exit. If the parameter is new, you will be asked for a parameter group to assign it to.[9] Parameter groups are described in section 2.4. The group should be a member of PARMGROUPS (or AUXPARMGROUPS if you are entering auxiliary information, see section 3.3 below). When prompted for a group, you need type only as much of the name as is needed to identify it; e.g. CUL for CUL-PARMS.

When a parameter is entered for the first time, you will be asked to classify it as "Yes/no", "Singlevalued", "Multivalued", or "Ask-all". Yes/no parameters have no EXPECT property, and take on the "values" YES and NO (true/false). Singlevalued parameters may take on any of several values, but the values are mutually exclusive; there may be competing hypotheses for its value, each with a CF less than 1.0, but never is it possible (both conceptually and in actual use) to have more than one value with certainty. Multivalued and Ask-all parameters may take on any number of values independently; see description of MULTIVALUED property for elaboration.

After you have classified the parameter, you will be prompted for all necessary properties (see below) for that classification. You may then additionally enter any optional properties you desire.

When prompted for a property (the prompt is "Property:"), you may enter the property name, or any of the following special responses:

DEL (or DELETE) – delete this parameter.
GROUP – change the group of the parameter (you will be prompted for new group).
NAME – change the name of the parameter (you will be prompted for new name).
STOP – abort this routine.
P – print the current properties of this parameter.
<cr> – to stop entering properties and go on to the next parameter.

Figure 3-1: Parameter Editor commands

---

[9] Should you enter a new parameter group, you will be asked for confirmation and to classify it as regular (in PARMGROUPS) or auxiliary (in AUXPARMGROUPS). However, it is unlikely you will ever need to create new groups in this fashion. You should, instead, define the new parameter group with respect to a new context type using the context acquisition routine.

If you specify a property name, control passes to the property editor, as described section 6.3.[10]

3.2. Properties of Parameters

Below is a list of the properties that a parameter can have. All parameters need a TRANS. If the value of the parameter is ever requested of the user, it needs a PROMPT and EXPECT, and if applicable, ASKFIRST. Numeric parameters usually have a RANGE property, and if appropriate a UNITS.

TRANS — How to translate the parameter. The TRANS is a list of words; if it contains an asterisk, the latter will be filled at output time with the translation of the context to which this parameter belongs; e.g., (the identity of *) might be translated as "the identity of ORGANISM-1". There can only be one asterisk. For yes/no parameters, any special verbs, such as "is", "has" (all those on the list TRANSVERBS), as well as the word "not", if present should be *lower-case* literal atoms for correct translation of the negation when the parameter is used in rules. If you enter the TRANS in all uppercase it will be converted to lowercase for you automatically.

PROMPT — How to ask for the parameter's value; no PROMPT means that it makes no sense (or is undesirable) to ask the user for the value. If the PROMPT is T, a prompt will be constructed from the TRANS. Otherwise, the PROMPT is a list; when the question is asked, the asterisk (at most one) in the list is replaced by the translation of the context being asked about. For multivalued parameters (except those that are ASK-ALL), the atom VALU is replaced by the particular value being asked about. To allow computation of parts of the prompt at runtime, PROMPT may contain one or more elements of the form (E <expr>); the result of having Lisp evaluate <expr> is spliced into the prompt.[11]

REPROMPT — More specific than the original prompt, the REPROMPT value is printed out when the user enters "?" in response to the original prompt or gives an incorrect response. In the case of singlevalued parameters, it has the same form as PROMPT, including an optional asterisk to refer to the context being asked about. This property is optional.

Parameters that are Multivalued may have a value-specific REPROMPT. In this case, the REPROMPT is an association list, one entry for each possible value, each of the form (value . response). The value NIL stands for any value not covered by any other entry; thus you can give specific reprompts for a few special values and have a more general reprompt for the remaining values.

EXPECT — The set of responses the client can use to answer questions about this parameter.

---

[10] It is possible to invoke the parameter editor to edit a well-defined collection of parameters. For example, you might want to edit the TRANS of everything in STRUCTURE-PARMS. To do this, you can type (to Lisp):
←(GETPARMS STRUCTURE-PARMS 'TRANS)
The first argument to GETPARMS is the list of parameters that you want to change, and the second argument is a property (or list of properties) to enter or edit. GETPARMS then uses successive elements of the list of parameters instead of prompting for a parameter, and for each such parameter allows you to enter or edit the desired property (or properties). Either argument may be NIL, in which case GETPARMS prompts for the omitted item.

[11] Both TRANS and PROMPT lists may contain quoted ("") strings, atoms, or a combination of quoted strings and atoms. The use of quoted strings saves some space but note that any special verbs for Yes/no parameters must be entered as atoms.

The EXPECT property is typically a list of the values (represented as atoms) or one of the following atomic values:

NIL — No EXPECT prop at all. Yes or No are the legal responses. [note: do *not* give yes/no parameters an EXPECT of (YES NO), as Yes/no parameters are treated specially. If you answer "Yes/no" to the classification question, this null EXPECT is implied.]

ANY — no restriction of the value.

NUMB — the value must be a number.

POSNUMB — the value be must a positive number.

FIXP — the value must be an integer.

DATE — the value is a date in the form mo/day/year (e.g. 3/30/79) or month day, year (e.g. MAR 30, 1979)

When EXPECT is a list, an element from it may be a list containing a single element instead of being an atom. In this case, that element is evaluated in order to produce a list of values. Most commonly the "code" to be *evaluated* is the name of a list, which is useful if the same list of possible values is to be shared by several parameters. The code to be evaluated, however, may be arbitrary Lisp code that produces a list. In this way, one can provide a context-dependent list of legal values; e.g., in MYCIN, valid answers for COLLECT (method of collecting a culture specimen) depends on the answer to the earlier question SITE. The expression to be evaluated may reference the free variable CNTXT, the object about which you are asking the question. Parameters with context-dependent EXPECTs *must* have a LEGALVALS property so that the system can determine (independent of context) *all* the legal values of the parameter. This is required for checking the legality of any values used in rules.

MULTIVALUED Indicates to the system that this parameter is multivalued, that it can have several different values with certainty (CF = 1.0) at the same time.

The value of the MULTIVALUED property may be T or ASK-ALL and is set when the system-builder classifies the parameter. The choice influences how the values of the parameter are requested. If MULTIVALUED = T, a separate question will be asked for each value that is needed in a rule. For example, the clause (SAME CNTXT ALLERGIC PENICILLIN) in a rule may cause the system to ask "Is the patient allergic to penicillin?". A question will be asked *only* if the value is needed in a rule. During a consultation, if no rule is tried that uses a given value, the system will not ask about that value. Since the expected answer to these questions is yes or no, the parameter needs no EXPECT (and, in fact, must *not* have one), but it should have a LEGALVALS property, which is the list of all possible values. The PROMPT for these parameters must contain the atom VALU; the value that appears in the rule will be substituted for this atom when the question is asked.

If MULTIVALUED = ASK-ALL, one question will be asked in which the user is expected to give all the values (e.g., for ALLERGIC, the question could be "Please list all the antibiotics to which the patient is allergic."). In this case, the parameter's EXPECT property must be the list of possible values (since these values will be entered in response to the question).

The TRANS of a multivalued parameter is by convention stated in the plural (e.g., (the drugs to which * is allergic)). This phrasing is necessary for proper translation throughout the system.

LEGALVALS This is always the list of all legal values for this parameter (as opposed to the EXPECT prop, the values which the user is EXPECTed to enter when asked a question about this parameter). This property is needed only for parameters with MULTIVALUED property T and for parameters whose EXPECT property contains context-dependent code. When a parameter has no LEGALVALS, the legal values are assumed to be specified by the parameter's EXPECT property. Note that a parameter with neither an EXPECT nor a LEGALVALS will be treated as a yes/no parameter; thus if you have a Multivalued parameter that has no EXPECT (because it is never prompted for), you need to give it a LEGALVALS.

The LEGALVALS property, like the EXPECT property, may be a list or an atom. Three atomic forms are recognized:

ANY The parameter may take on any value.

CNTXT The parameter takes context instances as its value For example, in MYCIN, the TREATFOR property of the patient is the list of all the organisms in the dynamic context tree that should be treated (e.g., ORGANISM-1, ORGANISM-2, etc.);

TEXT The parameter takes arbitrary pieces of text as its value; see CONCLUDETEXT in appendix I. This is commonly the case for some goal parameter—the text will be the system's final analysis or recommendations. TEXT is *never* a value for EXPECT, as it cannot be input—if you want to accept arbitrary value for input of a parameter, give it the EXPECT of ANY.

The forms POSNUMB, NUMB, FIXP, and DATE are applicable only to single-valued parameters are are never used for LEGALVALS.

ASKFIRST If T, the system will find out the value of this parameter by first asking the user before trying any rules. This property was originally called LABDATA and meant that the parameter was the result of a quantitative lab test; this has been generalized to be any piece of information that the user is likely to know. If the user does not answer the question, or gives an uncertain answer, the system will use rules (if any exist) to deduce the value. For parameters that have no ASKFIRST property, the system first tries to conclude the value using rules, and only asks if no value was concluded (and a PROMPT exists).

PROPERNOUN If the value of the parameter should be capitalized in translation (e.g., NAME), then the parameter should have PROPERNOUN property with value T.

UNITS For numeric-valued parameters (where the value of EXPECT is one of NUMB, POSNUMB, or FIXP) the default units, if any, in which it is measured. If the parameter has this property, the client can give the value in a different unit, and the system will convert it to the default units. Values in rules that use this parameter assume that the value is stored in these units, and the system translation functions will use these units when translating numeric quantities. The following units are permitted: CMS, DAYS, FEET, GMS, HOURS, INCHES, KGMS, METERS, MGS,

MINUTES, MONTHS, POUNDS, WEEKS, YARDS, YEARS. The singular forms, as well as the abbreviations FT, HRS, KG, LBS, MIN, YRS will be recognized in client input.

The following properties are optional. They improve the system's question-asking behavior, provide more complete information about a request, and allow a broader range of client responses to a question.

RANGE A list indicating the legal range of values of a numeric-valued parameter (one with EXPECT of FIXP, NUMB, or POSNUMB,) and what to do if the client enters a value outside the range. The first two elements of the list are the lower and upper bounds of the range, respectively. If the next element is the atom CONFIRM, then the client will be required to confirm that an out of bounds answer is really correct, otherwise the bad answer is unconditionally rejected.

The remaining elements of the list are optional and, if given, specify a prompt for confirmation (when CONFIRM is given) or a statement that the value is illegal. The prompt or statement may include the atom VALU to stand for the client's input value. If no prompt/denunciation is specified, the system generates something appropriate. For example, the parameter AGE in MYCIN has the following RANGE property:

```
(0 100 CONFIRM Is the patient really VALU years old?).
```

CHECK A form to EVAL to make sure that the user's response is "reasonable", above and beyond the test inherent in the parameter's EXPECT property. The form may use the free variable VALU to access the user's response (an atom). CHECK is a generalized form of the RANGE property, usable by any type of parameter. In particular, to introduce a new type of value not covered by the normal EXPECT types, one could give a parameter the EXPECT of ANY and write a suitable CHECK to verify that it is correct.

For example, in the DART system the parameter TP.CNTLR.ADDR has a CHECK:

```
(CHECK.HEX VALU 3).
```

The function CHECK.HEX was defined and used to check the length and legality of the hexadecimal number given by the client.

The following are for specialized applications.

SPECIAL Indicates ambiguous answers to the PROMPT . It is a list, one entry per answer. The entries are usually pairs of the form

```
(<ambiguous response> <request for clarification>).
```

An entry may also be a triple with third element that provides a default value to use if the user responds to the request with UNKNOWN.

XTRASPECIAL Like the SPECIAL prop, this property also handles unusual user responses to a question about the parameter. It is a list of entries, one per anticipated response. However, unlike SPECIAL, the user response is not ambiguous, but rather answers more than one question. Generally, an entry is of the form

(<response> (<parm1> <value1>) (<parm2> <value2>) ...).

If the user enters <response>, the system will conclude that parameter <parm1> has value <value1>, etc. In each entry, one of the (parm value) pairs must be for the parameter being asked (i.e., the parm that has this XTRASPECIAL property).

There are some cases where the anticipated answers cannot be conveniently enumerated, e.g., if the response is a number, or a similar action is to be taken for many possible answers. A piece of Lisp code is used to handle these cases. The form of the entry becomes

(<code> (<parm1> <parm2>)).

If <code> (a Lisp expression) evaluates to a non-NIL value, this value is the value for <parm1> and the user's response is the value for <parm2>. The code being evaluated can reference the free variables VALU (the user's response), PARM, and CNTXT. Again, either <parm1> or <parm2> must be the parameter that is being asked about (usually it will be <parm1>, the parm for which the code computes the value).

This form is commonly used to provide a "screening" yes/no parameter. This is a parameter that will be asked before a set of other questions is asked. If the answer is no, the remaining questions will not be asked. For example, in MYCIN, the system first asks if the results of a CBC (complete blood count) are available before asking for those results. When results are available, users often enter one of the results (the white blood count (WBC)) in answer to this question instead of YES. The XTRASPECIAL prop for CBC says that if a physician enters a number, pretend she meant to say "YES" and that the number is the answer for the WBC parameter.

LABEL.ORDER To be used when the LEGALVALS is the atom TEXT. The value of this property controls the order in which conclusions with the same CF are printed by PRINTCONCLUSIONS. To be of use, conclusions about the parameter must have been made by CONCLUDETEXT with a call to the function TEXT as the concluded value so that a label is associated with each value (see section I.5). If LABEL.ORDER has the value T, the labels are integers, and the concluded values will be sorted in ascending order of label; otherwise, the LABEL.ORDER property is a list of labels (atoms), and the values are sorted according to the order of the labels in this list. Labels that are also "text tags" must be members of the auxiliary parameter group TEXTAGS (see section 3.3).

UPDATE A value of T indicates that this parameter is likely to change over time. This property is used only with the UPDATE option (see section 8.2), and only with parameters that can be asked, that is, parameters with PROMPTs. This property indicates that the system should request an updated value for the parameter when a consultation is run using the UPDATE option.

Parameters additionally have a number of properties, generated automatically, that indicate which rules they are used in. These properties are: USED-BY, UPDATED-BY, CONTAINED-IN, ANTECEDENT-IN, UPDATED-IN, USED-BY-THE-WAY, and UPDATED-BY-THE-WAY. See section 9.2 for details.

3.3. Examples of Parameters

Below are some additional parameter definitions taken from the MYCIN system.

JUNKIE *a yes/no parameter of a PATIENT context*

```
TRANS:  (* has "a history of illicit intravenous drug use")
PROMPT:  (Does * "have a history of illicit intravenous drug use?")
ASKFIRST:  T
```

FEBRILE *a parameter of a PATIENT context*

```
TRANS:  (* is "febrile")
PROMPT:  (Is * "febrile?")
ASKFIRST:  T
UPDATE:  T
```

AGE *a single valued parameter of a PATIENT context*

```
TRANS:  (the age of *)
PROMPT:  (Age:)
ASKFIRST:  T
EXPECT:  NUMB
RANGE:  (0 100 CONFIRM Is the patient really VALU years old?)
UNITS:  YEARS
```

GRAM *a parameter of an ORGANISM context*

```
TRANS:  (the stain of *)
PROMPT:  (The stain (Gram, India ink, or Ziehl-Neelsen acid-fast)
      of * :)
REPROMPT:  (There are three principal stains used for examining smears
        of organisms.  Most bacteria stain Gram positive or
        Gram negative.  Mycobacteria are typified by acid-fast
        Ziel-Neelsen staining.  Cryptococcus, a fungus, is
        demonstrated by a positive India ink stain.)
ASKFIRST:  T
EXPECT:  (GRAMNEG GRAMPOS ACIDFAST INDIAINK)
PROPERNOUN:  T
```

ALLERGY *a multivalued parameter of a PATIENT context*

```
TRANS:  (the drugs to which * is allergic)
PROMPT:  (Is the patient allergic to VALU ?)
MULTIVALUED:  T
LEGALVALS:  ((ALLDRUGS))
```

SITE *a parameter of a CULTURE context*

```
TRANS:  (the site of *)
PROMPT: (From what site was the specimen for * taken?)
EXPECT:  ((ALLSITES))  That is, the value of the list called
                       ALLSITES contains the actual values
ASKFIRST:  T
SPECIAL:  ((ABSCESS (Please give the site of the abscess (e.g., LIVER,
                    LUNG-TISSUE, BRAIN, etc.))
          (KIDNEY (Do you mean URINE or RENAL-TISSUE ?))
          (LUNG (Do you mean SPUTUM or LUNG-TISSUE ?))
          (SINUS (Do you mean SURG-WOUND , TRAUMA-WOUND , or
                    CRANIAL-SINUS ?))
```

```
            (EAR (Do you mean MIDDLE-EAR or EXTERNAL-EAR ?))
            (PERIDURAL-SPACE (Please use either SUBDURAL-SPACE or
                                        EPIDURAL-SPACE))
            (SKIN (Do you mean SUBCUTANEOUS , CUTANEOUS-ULCER ,
                      BURN-SITE , SKIN-SURFACE , NAIL , IV-SITE ,
                      SURG-WOUND , or TRAUMA-WOUND ?))
            (EYE (Do you mean CONJUNCTIVAE , CORNEA , ANTERIOR-CHAMBER
                      , or VITREOUS ?)))
  XTRASPECIAL:  ((TRANSTRACHEAL-ASPIRATION (SITE SPUTUM)
                              (COLLECT TRANSTRACHEAL-ASPIRATION)))
```

CBC *a parameter of a PATIENT context*

```
  TRANS:  (the results of a recent CBC of * are available)
  PROMPT:  (Do you know the results of a recent CBC of * ?)
  ASKFIRST:  T
  XTRASPECIAL:  (((AND (NUMBERP VALU)
                       'YES)
                  (CBC WBC)))
```

[Note: this means that if the user enters a number, the answer to the question asked is "yes", and the number is the value for the parameter WBC.]

Auxiliary Parameter Groups

The list AUXPARMGROUPS defines "auxiliary" parameters. These are not really parameters in the strict sense, that is, parameters of contexts. Rather they are structures that are used for various purposes during a consultation and are conveniently acquired, stored and manipulated in the same manner as true parameters. You may never need to manipulate them explicitly in your early system building. AUXPARMGROUPS initially contains:

RULEGROUPS	The list of rule groups.
FUNCTIONS	The list of user-defined functions to be used in rules.
DOMAIN.VARIABLES	The list of domain-specific atoms that will need TRANSes or other parameter properties for use in user-defined functions. For example, lists of values to be used in rule premise clauses, see the function LISTOF.
TABLES	The list of simple blocks of parameters for use in tabular input (see section 8.3).
TEXTAGS	The list of text "tags" which are used in creating values for TEXT-valued parameters (functions that use text tags are described in section 4).

4. Rules

The largest component of the knowledge base for an EMYCIN consultant is the *rule base*—a collection of production rules that instruct the computer about how to make inferences and arrive at conclusions for the values of the parameters in the context tree. The rules capture the expert's problem-solving methods and judgmental rules of inference. They determine the ordering of the the goals and subgoals to be explored during a consultation.

The rules are expressed in the vocabulary of contexts and parameters described in the preceding sections (2 and 3). Thus, while rules describe the *action* or *problem-solving component* of the expert's knowledge, the contexts and parameters record the *structural* information about the domain. This chapter will describe the internal structure and entry of individual EMYCIN rules, describe the operation of the rule interpreter, then briefly sketch some of the ways the rules have been employed in existing consultants. Additional details about the rule entry using the Abbreviated Rule Language (ARL) occupy the remaining sections of this chapter.

4.1. The Structure and Definition of Rules

Like parameters and context types, each EMYCIN production rule has several constituent properties. In particular, each rule must have a PREMISE property and an ACTION property. The PREMISE and ACTION properties have a stereotypic internal format and refer to predicate and action functions (see appendix I), the domain's parameters, and their values.

```
RULE010

If:  1) The ptt of the patient is normal,
     2) The pt of the patient is high,
     3) The tt of the patient is normal,
     4) The bleeding time of the patient is normal,
     5) The platelet count of the patient is normal, and
     6) The urea solubility test of the patient is normal
Then:  It is definite (1.0) that the following is one of the coagulation
          defects of the patient: factor VII deficiency

AUTHOR: CSD.BENNETT
LITERATURE: Harrison, Handbook of Internal Medicine
JUSTIFICATION: Indication of extrinsic enzyme pathway defect
CREATED:  2-Nov-80 17:42
PREMISE: ($AND (SAME CNTXT PTT NORMAL)
               (SAME CNTXT PT HIGH)
               (SAME CNTXT TT NORMAL)
               (SAME CNTXT BT NORMAL)
               (SAME CNTXT PC NORMAL)
               (SAME CNTXT FSF NORMAL))
ACTION: (CONCLUDE CNTXT COAG-DEF FACTORVII TALLY 1000)
```

Figure 4-1: A sample rule from the CLOT system.

A rule's PREMISE is always a call to the function $AND of the form ($AND $clause_1$ $clause_2$ ...

clause$_n$). Each clause typically has the form (<function> CNTXT <parameter> <value>). The ACTION of a rule is either a call to a single conclusion function or, if there are multiple conclusions, it has the form (DO-ALL conclusion$_1$ conclusion$_2$ ... conclusion$_n$). The premise and action clauses are evaluated in their order in the $AND or DO-ALL functions.

Rules and their properties may be entered or modified using the Rule editor, which can be called from the EMYCIN executive (the "Rules" command, see section 6.2). It prompts for a rule number (to revise old rules), or NEW to enter a new rule. For a new rule, it prompts for the body of the rule (PREMISE and ACTION parts); for an old rule, it asks you what property of the rule you want to change. After you have entered the rule, the editor checks the rule for syntactic interactions with other rules. You may exit the Rule editor by typing <cr> in response to the main rule prompt.

When you are prompted to enter a property name to change, you can enter a property name (e.g., PREMISE, ACTION), one of several commands to translate the rule (e.g., TRANSLATE for English, PP for prettyprinted Lisp, and PPT for ARL or "terse" form), or commands to delete or check the rule. The PREMISE and ACTION properties are entered using the property editor (see Figure 6-2), which allow you to enter a value for the property or modify any existing value. Typing a <cr> to this part of the routine stops editing the current rule.

Antecedent Rules

Most of a system's rules are "consequent" rules, to be used in a goal-directed manner. These rules will be invoked in order to find out the value of a parameter in their ACTION. However, there are occasions when the expert wants to take an action or make a conclusion about a parameter whenever some other fact or situation occurs. EMYCIN supports a limited "antecedent rule" facility to handle these situations.

Antecedent rules are applied when all of the information in their premise becomes known, either by answers to questions or by the conclusions of other rules. However, antecedent rules never cause questions to be asked, cause context instances to be created, or cause consequent rules to be applied. Furthermore, antecedent rules can only be "chained forward" when they conclude the value of a parameter with certainty. In such a case, any antecedent rules for the concluded parameter are also applied.

A principal use for antecedent rules is to make definitional conclusions that one would not want to make via the normal backward-chained deductive mechanism. Concluding the value of a parameter with certainty prevents the asking or inferring of the parameter later in the consultation. Thus antecedent rules can be used to "out-guess" answers the client might be asked to provide (eliminating a question). This is especially true if the parameter is normally asked before any consequent rules could be applied to make this guess. Antecedent rules can also be used to "shortcut" or eliminate the application of consequent rules (reducing the amount of deduction the system has to perform).

A typical antecedent rule is shown below. Ordinarily, the MYCIN parameter *the patient has had a genito-urinary manipulative procedure* would be asked during the consulation, However, if the user has already told the system that a urine culture was obtained by one of several methods of collection,

this question would be redundant. The following rule is triggered when the method of collection of the urine culture is found out, thereby "answering" this subsequent question.

RULE091

```
If:  1) The site of the culture is urine, and
     2) The method of collection of the culture is one of: foley-cath-
        aspiration foley-bag-drainage straight-catheterization ureteral
Then:  It is definite (1.0) that the patient has had a genito-urinary
         manipulative procedure
```

To make a rule an antecedent rule, you must set its ANTECEDENT property. If the rule is currently an antecedent rule, modifying the ANTECEDENT property will cause the rule to become a consequent rule.

Information Properties

The following (optional) properties organize justification and design information about rules that should be recorded in an knowledge base. This information is invaluable especially if the knowledge base is intended to tutor the client and if other experts are contributing to the knowledge base.

COMMENTS	The COMMENTS of a rule are any special remarks indicating why the rule is included in the rule base: what special case does it cover, how might it be improved, what other rules is it related to, etc. Imagine that this rule is shown to a student, what do you want him/her to know about the pros and cons of this rule?
JUSTIFICATION	The JUSTIFICATION of a rule describes why the rule is true. In general, the justification should be a detailed description of why the rule makes sense, including the rationale for each clause of the rule (when this is not obvious). Include any information that might be useful to a student who is trying to learn the rule.
LITERATURE	Any literature references that support the justification of this rule.
CASES	The cases that exemplify the use of this rule.

The remaining rule properties, AUTHOR, DATE, and LASTEDIT, have their values maintained automatically by EMYCIN.

4.2. The Rule Interpreter

Rules in the rule base are applied to a case by the rule interpreter, called FINDOUT. The task of FINDOUT is to establish the value of a parameter with respect to a specific context instance. This process of establishing a parameter's value is called *tracing*. Parameters for FINDOUT to trace are specified in the GOALS and INITIALDATA lists of context types and in rule premise clauses.

The strategy that the interpreter follows to trace a parameter depends on the parameter's definition (see section 3). If the parameter is an ASKFIRST parameter and it has a PROMPT property defined, FINDOUT uses the PROMPT to first ask the client for the answer. If the parameter is still not known with certainty after the client's answer, then the interpreter retrieves a precomputed list of rules that can make conclusions about this parameter. Each of the rules in the list that has an applicable

SUBJECT (see the next section) are evaluated with respect to the current context instance (and its ancestors). Evaluating a rule may cause other parameters to be traced as subgoals.

If the parameter is still unknown after all of the rules have been evaluated, the system attempts to ask the client for values, assuming, of course, that it has not already done so and the parameter has a PROMPT property defined. Thus, if the parameter does not have a PROMPT property defined, the system will never ask the client to supply values. Parameters that serve as the main goals of the system are normally defined with no PROMPT.

Following this process the parameter is considered traced in this particular context instance and it will never be reconsidered during this consultation. Any antecedent rules that might be triggered by this parameter are then evaluated.

The rule interpreter has several additional mechanisms that modify its operation. If the value of a single-valued parameter is concluded with certainty then no further rules need be tried. Thus, before the parameter is traced in the normal way the system attempts to find a rule that would establish the parameter with certainty, based only on what is currently known. The selected rule is termed a Unitypath to finding the value of the parameter. When such a rule is found for an ASKFIRST parameter, it saves asking the client a question; when one is found for ordinary parameters, it may have saved needlessly tracing some other parameters.

Another optimization is based on the fact that if any clause of a rule's conjunction of premise clauses is false, then the entire rule will fail. FINDOUT "previews" all of the premise clauses to see if one is already known to be false in this context. If such a clause is found the rule is not tried at all and any remaining, perhaps unknown, clauses that could cause the system to create additional subgoals are not evaluated, thereby avoiding futile exploration of a subproblem.

EMYCIN allows a rule to mention the same parameter in both its premise and action. In order to prevent these so-called *self-referencing* rules from causing an infinite loop (by attempting to trace the parameter whose tracing invoked the rule in the first place), the rule interpreter treats these rules as a separate rule set. FINDOUT first applies the set of non-self-referencing rules in the usual fashion. It then applies the self-referencing rules, locally evaluating all references to the common parameter in terms of the (constant) value established by the non-self-referencing rules.

The creation of context instances

One final aspect of the FINDOUT mechanism needs discussion: the creation of context instances for a context type and the propagation of the context tree itself. Recall that a rule premise clause does not refer to specific instances of contexts. In fact, each clause refers (through the parameters mentioned in the premise clauses) only indirectly to the context types that are required to have instances during the consultation. As it evaluates each premise clause, FINDOUT determines the context types the clause refers to and insures that *all* the instances of this type (if any) are created before attempting to apply the current rule clause to any of them.

A consultation about a case begins with the automatic creation of a single instance of the root context type and having FINDOUT trace its INITIALDATA and GOALS parameters with respect to this

instance. In the process of tracing these parameters rules may be evaluated that refer to a parameter of another context type that has not yet been created. In this case, FINDOUT suspends the tracing of this parameter until it has created all the instances of this (and any intervening) context type.

Creation of context instances is controlled by the definition of the context type. FINDOUT uses the definitions of the PROMPTEVER, PROMPT1ST, and PROMPT2ND properties to interact with the client about the existence of an initial and perhaps additional instance of this type. When each instance is created its INITIALDATA and GOALS parameters (if any) are traced with respect to this new instance. Finally, after all the instances have been created (as, for example, when the client indicates that there are no more instances of this context type) then the original parameter that triggered this mechanism is traced with respect to all the newly created instances.

Rule evaluation

Evaluation of an individual rule clause occurs in a manner similar to Lisp's EVAL function, but differs on the handling of unquoted atoms. If an argument to a predicate or action function is a list it is treated as a function call. The clause is evaluated and the resulting value is passed as the argument to the calling function. Non-lists are taken literally (e.g., HIGH, PTT, COAG-DEF, and FACTORVII in figure 4-1), except those atoms that are designated as variables, which are replaced by their values. If you want a list to be taken literally (not interpreted as a function call), you must QUOTE it.

If you want a non-list to be evaluated, you must mark it as a variable or use the function LISTOF (see section I.4). Atoms that are to be used as variables are indicated by giving them an ALWAYSEVAL property (using the Variables command). Currently the atoms CNTXT, TALLY, and the variables used in mapping functions (FREEVAR, COLLECTEDLST) have this property.

The atom CNTXT used in the premise and action clauses of this rule is a variable bound by the rule interpreter during the consultation to the current context instance. When a rule is applied to a context instance the system uses the value of this variable to look up the values of the parameters with respect to this instance and any ancestor context instances.

Computing Certainty Factors

EMYCIN systems must often deal with domains in which reasoning is judgmental and inexact. To represent this aspect of domain knowledge, the certainty factor mechanism allows the expert to make such statements as "having been burned seriously suggests that Pseudomonas is a possible organism causing the infection" and "a normal prothrombin time tends to rule out hemophilia". EMYCIN provides a method for computing and combining certainty factors that are used in rules.

When a rule premise (such as the example rule shown in figure 4-1 above) is evaluated, each clause within it returns a certainty factor if true (or T, which implies CF = 1.0) and NIL if false. The function $AND will set the value of the variable TALLY to be the minimum of the certainty factors returned by each of the premise clauses. If TALLY is greater than .2 the premise is considered satisfied and the action is evaluated. In this case, the function CONCLUDE sets the certainty factor of the indicated triple, in this case (CNTXT COAG-DEF FACTORVII), to be TALLY multiplied by the

certainty factor of the rule (1.0).

If this concluded triple had already been concluded by another rule then the system combines the certainty factor of the previous conclusion, CF_p, with the newly-computed certainty factor, CF_n, to determine the triple's cumulative certainty factor. This is accomplished by the function CFCOMBINE, which is currently defined as follows:

$CFCOMBINE(CF_p\ CF_n) =$

$CF_p + CF_n\ (1 - CF_p)$	*if CF_p and CF_n are positive.*
$(CF_p + CF_n) / (1 - \min\{\|CF_p\|, \|CF_n\|\})$	*if $CF_p CF_n < 0$*
$-CFCOMBINE(-CF_p, -CF_n)$	*if CF_p and CF_n are negative.*

Figure 4-2: CFCOMBINE

4.3. Rule Groups

This section describes details of the rule group mechanism. Unless you have defined context types that share a parameter group (see section 2.4), you should skip this section.

When a new rule is entered, it will be assigned to a *rule group*; this group is called the rule's SUBJECT. The SUBJECT of a rule determines the of context types to which the rule can apply. Each simple rule group represents a path in the static context tree from some node back to the root; rules in the group mention only parameters of the context types on that path. Compound rule groups represent the union of simple parameter groups. The gobal variable RULEGROUPS enumerates all the groups that have been defined (either automatically by the system or by the system builder).

Unless you have context types that share parameter groups, RULEGROUPS will contain only simple groups. The SUBJECT of every valid rule is then uniquely determined, and simply indicates the lowest node in the context tree that parameters mentioned in the rule belong to. In this case you will never need to manipulate the rule's SUBJECT yourself.

The context tree for the MYCIN system provides examples of valid rule groups.

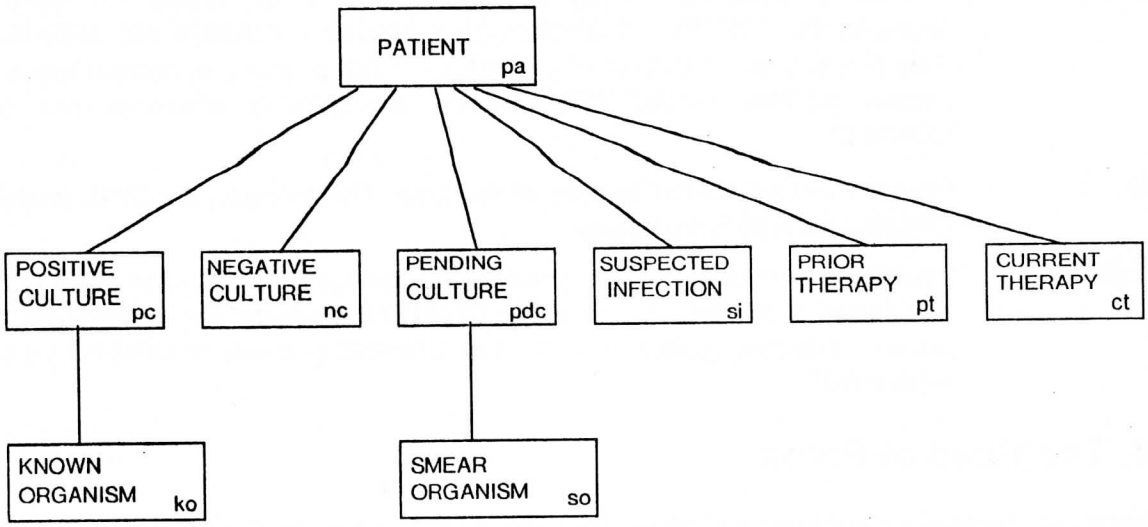

Figure 4-3: MYCIN's static tree

The simple rule groups for the MYCIN system are PATIENTRULES, which mention only parameters of the patient (pt) context type; POSCULRULES, which mention parameters of positive cultures and the patient (pt and pc); KNOWNORGRULES (pt, pc, and ko); NEGCULRULES (pt and nc); PENDCULRULES (pt and pdc); SMEARORGRULES (pt, pdc, so); SUSPINFRULES (pt and si); PRIORTHERRULES (pt and pd); and CURTHERRULES (pt and ct).

The context acquisition routine automatically defines, for each new context type, a rule group that applies to that context type (and its ancestors). However, if several context types share a parameter group , rules that use these shared parameters will have an ambiguous SUBJECT and you will be asked to resolve this ambiguity at the time that you enter the rule. For example, in the case of the MYCIN system, positive, negative, and pending cultures and suspected infections all share the parameter group PROP-CUL. Thus any rule that uses a parameter from this parameter group could be a member of POSCULRULES, NEGCULRULES, PENDCULRULES, or SUSPINFRULES.

If several context types do share a parameter group, it is possible to define "compound" rule groups that apply to multiple context types. You must define these rule groups yourself, either by assigning the new group as the rule's SUBJECT when the rule is defined (using the Rule editor) or by defining the rule group directly using the Parameter editor. In the later case, you must give the rule group a name and assign it to the Group RULEGROUPS.

In addition to the simple groups listed above, the MYCIN system contains the following compound rule groups:

CULRULES	apply to all "cultures" (pt, pc, nc, pdc, and si),
REALCULRULES	apply to "real" cultures (pt, pc, nc, and pdc),
ORGRULES	apply to all organisms (pt, pc, ko, pdc, and so),
THERRULES	apply to all "therapies" (pt, pd, and ct),

Each of these special rule groups needs the following properties; they are prompted for automatically when you try to assign a rule to the new subject.

CONTEXT — The list of lowest context types to which rules in this group can apply. For example, the CONTEXT of ORGRULES in MYCIN is (KNOWNORG SMEARORG). This property is the inverse of the RULETYPES property of context types. The system modifies RULETYPES for you appropriately whenever you modify CONTEXT.

SVAL — How to translate CNTXT in rules of this type. For example, the SVAL property of ORGRULES is (this organism).

CTRANS — An English phrase (string) describing what context types the rules apply to (i.e., a translation of the information in the CONTEXT property; fills in the blank in the phrase "this rule applies to —"). The CTRANS property of ORGRULES is "all organisms".

4.4. The Uses of Rules

As with parameters and contexts, rules have been employed in a variety of ways in existing EMYCIN systems. This section describes a few of the more stereotypic uses. Although these uses by no means exhaust the possible applications of rules in EMYCIN systems, they are indicative of the kinds and representation of knowledge that an expert might use when reasoning about a case.

Generally, rules have served to encode reasoning methods for

- **classifying case data** in a symbolic fashion,
- **identifying problems and their causes** based on simplified event scenarios,
- **making default inferences** in the absence of information or in the presence of conflicting information, and
- **ordering the tasks** that the consultant is to pursue.

In the following paragraphs we present examples of each of these kinds of expertise using rules taken from several EMYCIN systems.

Classifying Case Data

A considerable amount of inference that a consultant performs is classificatory in nature: determining how situations or data can be meaningfully grouped or clustered. In each of the existing EMYCIN systems, the expert has always imposed one or more classification schemes on the case data that will be supplied by the client; rules that implement these classification methods thus form a substantial part of EMYCIN rule-bases. Consider the rule given below, an example taken from the MYCIN system.

RULE334

```
If:  1) It is not known whether the organism was able to grow
        aerobically,
     2) A: The site of the culture is blood, or
        B: The lab has attempted to grow the organism anaerobically, and
     3) The organism was able to grow anaerobically
Then:  There is evidence that the aerobicity of the organism is facul
       (.5) or anaerobic (.2)

PREMISE: ($AND (NOTKNOWN CNTXT AIRGROW)
               ($OR (SAME CNTXT SITE BLOOD)
                    (SAME CNTXT ANTRY))
               (SAME CNTXT ANGROW))
ACTION:  (CONCLUDE* CNTXT AIR TALLY '((FACUL 500)(ANAEROBIC 200)))
```

This rule addresses the situation in which a sample of the patient's blood is tested for the presence of certain kinds of organisms. The rule concludes about one feature of these organisms: their "aerobicity", that is, their requirement of oxygen for growth. This classification, in turn, is used when the consultant attempts to identify specific organisms that might be causing the patient's infection.

Rules of this form capture domain principles that classify laboratory data. MYCIN applies several additional such classification principles to the medical evidence in order to establish the genus and identity of the organism, to determine the general type of the infection based on the kind of organism that caused the infection, and to determine whether a particular infection or organism is significant. Finally, these classifications are used to identify the organisms that require therapy and to determine the specific drugs to be given to the patient.

The use of classification principles are not limited to medical domains. The SACON system, for example, classifies the response behaviors that a large structure will experience under certain operating conditions. These classifications eventually help determine what type of structural analysis the engineer should perform for the structure. One rule that performs such a classification is shown below.

RULE050

```
If:  1) The material composing the sub-structure is one of: the metals,
     2) The tolerable analysis error (in percent) is between 5 and 30,
     3) The non-dimensional stress of the sub-structure is greater than .9,
     4) The number of loading cycles to be applied is between 1000 and 10000
Then: It is definite (1.0) that fatigue is one of the stress behavior
        phenomena in the sub-structure

PREMISE: ($AND (SAME CNTXT MATERIAL (LISTOF METALS))
               (BETWEEN* CNTXT ERROR 5 30)
               (GREATERP* CNTXT ND-STRESS .9)
               (BETWEEN* CNTXT CYCLES 1000 10000))
ACTION: (CONCLUDE CNTXT SS-STRESS FATIGUE TALLY 1000)
```

Reasoning using classification principles allows rules to be stated succinctly. Furthermore, the terms used to express these classification principles in the system's rules form a common ground that

is shared by expert and client. Rules employing these terms are more easily comprehended and make explanations of the system's reasoning more transparent.

Reasoning based on Event Scenarios

A second type of expertise that has been represented in EMYCIN rules captures reasoning based on simple event scenarios. The premises of these rules attempt to establish that a particular outcome of a process has been observed and that some related, past event has also occurred. This past event might, for example, be required to enable to process to occur. The rule then concludes, on the basis of a common domain-specific scenario, that some other important event or object must also have participated in this case. The MYCIN system provides numerous examples of this type of reasoning. Several rules capture aspects of the infectious disease process in the human body and are used to conclude about organisms that are likely to cause a specific infection. The following rule is a striking example taken from the portion of the rule base that deals with a meningitis patient.

RULE578

```
If:  1) The infection which requires therapy is meningitis,
     2) A: A smear of the culture was not examined, or
        B: Organisms were not seen on the stain of the culture,
     3) The type of the infection is bacterial, and
     4) The patient has been seriously burned
Then: There is suggestive evidence (5.) that pseudomonas-aeruginosa is
        one of the organisms which might be causing the infection

PREMISE: ($AND (SAME CNTXT TREATINF MENINGITIS)
               ($OR (NOTSAME CNTXT EXAMSTAIN)
                    (NOTSAME CNTXT SPECSTAIN))
               (SAME CNTXT TYPE BACTERIAL)
               (SAME CNTXT BURNED))
ACTION: (CONCLUDE CNTXT COVERFOR PSEUDOMONAS-AERUGINOSA TALLY 500)
```

In this rule the effect of the disease process is known (the patient has bacterial meningitis) but the causal organisms have not been directly observed, either by the physician or by the laboratory. In addition the patient is known to be seriously burned. This allows MYCIN to conclude that a possible organism causing the infection is *Pseudomonas aeruginosa*. The following justification for this inference was provided by the expert: "For a very brief period of time after a severe burn, the surface of the wound is sterile. Shortly thereafter, the area becomes colonized by a mixed flora in which Gram-positive organisms predominate. By the third post-burn day this bacterial population becomes dominated by Gram-negative organisms. By the fifth day these organisms have invaded well beneath the surface of the burn. The organisms most commonly isolated from burn patients include *Pseudomonas aeruginosa*, which is frequently fatal."

The rule captures the idea that this particular organism can enter the body through the burn. This event then predisposes the patient to infections caused by the organism, in this case, meningitis. Exposure to organisms, in conjunction with other predisposing factors such as immunosuppression, permit a stereotypic disease process to occur in patients. There are several such rules in the MYCIN knowledge base that reason about the mechanisms of exposure to certain organisms.

Rules that reason about event scenarios are important to consultants that attempt to diagnose problems and their causes. For example, they form a large part of the knowledge base used in the DART consultant, which diagnoses computer component failures. The following (abbreviated) rule is taken from that consultant and makes use of knowledge about various steps in a telecommunications protocol to diagnose a failed attempt to activate a computer teleprocessing network. Knowing what step in the protocol did not occur allows the consultant to identify specific network components causing the problem.

RULE128

```
If:  1) The symptom for the physical path is failed-network-activation,
     2) VTAM leased line SDLC protocol is used to communicate with this
        terminal,
     3) Line is not one of the types of VTAM traces available for use on
        this problem,
     4) The VTAM IO trace shows an ACTLINK was sent to the line for this
        terminal, and
     5) The ACTLINK response shown in the VTAM IO trace is negative
Then:  There is evidence that the following components may be involved
       in this problem: line (.5) tp.cntlr (.2) tp.cntlrpgm (.2)

PREMISE:  ($AND (SAME CNTXT ACTUAL.SYMPTOM NETWORK-ACTIVATION)
                (SAME CNTXT VTAM.LEASED.SDLC)
                (NOTSAME* CNTXT VTAMSNA.TRACE LINE)
                (SAME CNTXT RNIO.ACTLINK)
                (SAME CNTXT RNIO.ACTLINK.RESP NEGATIVE))
ACTION:  (CONCLUDE* CNTXT PP.CAUSE TALLY
                    '((LINE 500)(TP.CNTLR 200)(TP.CNTLRPGM 200)))
```

Reasoning about Defaults

Another use of the rules has been to give the consultant some ability to reason in those cases where it is unable to conclude about some aspect of a case (or when it has several, possibly inconsistent or confusing answers about an aspect of a case). The expert is often able to provide a default conclusion or, in the case of confusing answers, is able to "second-guess" the situation. The next example rule illustrates the use of default reasoning in the MYCIN system. Notice that this rule is an example of a self-referencing rule, a rule that mentions a parameter in both its premise and action. Rules that perform this form of second-guess reasoning always use this method of representation.

RULE086

```
If:  1) The aerobicity of the organism is not known, and
     2) The time since the specimen for the culture was obtained is
        less than 2 days
Then:  There is evidence that the aerobicity of the organism is
         obligate-aerob (.5) or facul (.2)

PREMISE: ($AND (NOTKNOWN CNTXT AIR)
               (LESSP* (VAL1 CNTXT WHENCUL) 2 DAYS))
ACTION: (CONCLUDE* CNTXT AIR TALLY
                   '((OBLIGATE-AEROB 500)(FACUL 200)))
```

This rule will apply to a case when other rules, such as RULE334 above, are unable to conclude directly about the aerobicity of the organism. In this situation, since the organism grew out in a relatively short time, the consultant can conclude that the organisms might be either obligate aerobic or facultative. The other possibility, that the organism is anaerobic, is not indicated since such organisms take much longer to grow.

Controlling the Ordering of Tasks

A final but important use of the rules has been to order the tasks the consultant performs during a case. Rules in this class have been referred to as "goal rules" since they control the order in which the goals and sub-goals of the system are pursued.

Recall from the section dealing with contexts that there are two lists of parameters associated with each context type, INITIALDATA and GOALS. EMYCIN attempts to trace these parameters whenever an instance of a context is created during a consultation. These parameter lists specify the order that some of the goals of the consultant will be established—they will be pursued according to their order in the list. In addition to this mechanism, subgoals generated during the evaluation of a rule's premise are pursued according to the clause order of that rule. Thus it is possible to write rules that encode ordering information as well. These rules allow particular subgoals to be established depending on other case data, a desirable feature not possible using the simple INITIALDATA and GOALS mechanism.

The following example goal rule was taken from the CLOT system and demonstrates these points.

RULE001

```
If:   1) An attempt has been made to deduce the blood defects (based on
         clinical evidence only),
      2) An attempt has been made to deduce the blood defects (based on
         the coagulation battery only),
      3) There is a consistent, overall estimation of the blood defect
         of the patient,
      4) An attempt has been made to deduce the final blood disorders of
         the patient, and
      5) An attempt has been made to deduce the blood coagulation diagnoses
         of the patient
Then: 1) It is definite (1.0) that there is an interpretation of a
         bleeding defect in the patient,
      2) Display the final blood disorders of the patient,
      3) Display the blood coagulation diagnoses of the patient

PREMISE:  ($AND (ONCEKNOWN CNTXT CLINDEF T)
                (ONCEKNOWN CNTXT LABDEF T)
                (SAME CNTXT CONSISTENT)
                (ONCEKNOWN CNTXT FINALDEF T)
                (ONCEKNOWN CNTXT DX T))
ACTION:   (DO-ALL (CONCLUDE CNTXT INTERPRETATION YES TALLY 1000)
                (PRINTCONCLUSIONS CNTXT FINALDEF T T)
                (PRINTCONCLUSIONS CNTXT DX T T))
```

The first two clauses cause the consultant to gather certain kinds of evidence and to infer the

possible bleeding disorders of the patient based on each type of evidence independently. In the third clause, CLOT considers whether these independent estimations of the patient's problem are consistent. If they are, the consultant continues, finally establishing the diagnoses for the patient. Another, similar goal rule handles the situation in which the estimations are inconsistent.

These broad types of reasoning expertise comprise the bulk of the rules found in current EMYCIN systems. Although there are numerous alternative syntactic forms available for expressing this expertise, it is quite prudent to begin encoding reasoning expertise along the lines suggested by the examples above. Only after you become familiar with the interactions between the component parts of a knowledge base and the rule interpreter is it advisable to explore alternative representations of this or different, more complex expertise.

4.5. Abbreviated Rule Language (ARL)

In addition to entering the rules directly in the Lisp form, they may be entered in an abbreviated or "terse" format called ARL for Abbreviated Rule Language. This is an Algol-like language in which the operands are the parameters of the domain, and the operators represent EMYCIN predicate functions. As an example, the rule premise

```
($AND (SAME CNTXT SITE BLOOD)
      (GREATERP* (VAL1 CNTXT SICKDEGREE) 2)
      ($OR (NOTSAME CNTXT BURNED)
           (LESSEQ* (PLUS (VAL1 CNTXT NUMCULS)
                          (VAL1 CNTXT NUMPOS))
                    3)))
```

might have been entered as

```
(SITE = BLOOD AND
 SICKDEGREE > 2 AND
 ~BURNED OR NUMCULS+NUMPOS LE 3)
```

and could have been entered on a single line. Ordinary Lisp forms may be used wherever a parameter or value may appear; conversely, an ARL form (parenthesized) may appear as a component of a regular Lisp form. For example, the above could also have been written

```
($AND (SITE = BLOOD)
      (SICKDEGREE > 2)
      ($OR (NOTSAME CNTXT BURNED)
           (NUMCULS+(VAL1 CNTXT NUMPOS) LE 3)))
```

or many other ways. Clauses such as

```
(STRUCTURE = (LISTOF METALS))
```

are also accepted. Mentioning a yes/no parameter by itself translates using SAME and "~" prefacing a yes/no parameter translates using NOTSAME. For example,

```
(BURNED AND ~ALCOHOLIC)
```

is translated as

```
($AND (SAME CNTXT BURNED)
      (NOTSAME CNTXT ALCOHOLIC)).
```

The operator "," is also recognized as "AND". Thus you could say

```
( SITE=BLOOD, SICKDEGREE > 2, ~BURNED OR NUMCULS+NUMPOS LE 3 ).
```

In all forms that use a context, the variable CNTXT is assumed; there is no way in ARL to specify the iteration variable in a mapping clause. A numeric-valued parameter <numparm> used as operand in an arithmetic expression is translated as (VAL1 CNTXT <numparm>). If a numeric expression involves only one parameter whose UNITS are defined, then any numbers in the same expression can be qualified with a unit of measure; for example, if the parameter AGE has UNITS of YEARS;

AGE > 9 MONTHS,

would be converted to (GREATERP* (VAL1 CNTXT AGE) .75).

ARL can also be used in the ACTION of a rule, though the possibilities are more limited: only CONCLUDE-type forms are translated. The following forms are understood:

```
ACTION: (IDENT=E.COLI (.3))
ACTION: (IDENT ~= E.COLI (.5))              [same as IDENT = E.COLI (-.5)]
ACTION: (IDENT = E.COLI (.5) KLEBSIELLA (.2) PSEUDOMONAS (-3))
                                            [translates into a CONCLUDE*]
ACTION: (ALCOHOLIC=NO (.3))                 [same as ALCOHOLIC = YES (-.3)]
ACTION: (IDENTITY = TABLE)                  [the system will interact to construct a table]
```

In conclusions, the operator " = " has several possible meanings. The most common, in "<parm> = <value>" is for the function CONCLUDE. If the parameter is text-valued, the function is CONCLUDETEXT, and the <value> is taken to be a call to TEXT. Other cases are shown below. If a *cf* for a conclusion is omitted, the *cf* is taken to be 1.0 (definite).

Caveat: Although the format described here is fairly flexible with respect to mixing Lisp and ARL forms, the parser does not have very good error recovery if you stray too far from the syntax; every binary operator is expected to appear in the form <operand> <operator> <operand>, where the <operand>s are other ARL forms or regular Lisp forms. Be careful not to forget the ANDs (or commas) that separate premise clauses.

The complete list of operators is given below. In general, a single-character operator (= + * , etc.) may appear embedded in a large atom or separated by spaces from its operands. The exception is " – ", which is often used as a "space" character inside parameter and value names, and therefore will be ignored if not delimited by blanks. If you have other such character operators that appear inside parameter or value names, use the Variables command to add them to the list DONTUNPACKCHARS in order to avoid problems.

The operators are given below in decreasing order of precedence, i.e., in the absence of parenthesis-grouping, the order in which operators will be evaluated. For example, the expression x+y*z will be translated as x+(y*z), not (x+y)*z). Operators in the same group have equal precedence under left-to-right evaluation. The only non-standard precedence is that of AND and OR, which is reversed (because $AND always appears at the top level of a rule).

```
  Operator(s)  Translates into

(1)  - (unary)  MINUS

(2)  ↑          EXPT

(3)  *          TIMES
     /          QUOTIENT

(4)  +          PLUS
     -          DIFFERENCE

(5)  <   LT     LESSP*
     >   GT     GREATERP*
     =>  GE     GREATEQ*
     <=  LE     LESSEQ*
     =          SAME in PREMISE, CONCLUDE in ACTION
    ~=          NOTSAME in PREMISE, negative CONCLUDE in ACTION

(6)  ~ (unary)  NOT, or NOTSAME for yes/no parms

(7)  OR         $OR
(8)  AND  ,     $AND in PREMISE, DO-ALL in ACTION
```

Figure 4-4: ARL Operators

The arithmetic operators (groups 1 thru 4) apply to numeric quantities. Hence they would never appear as arguments to AND or OR, as the predicate operators would.

The following table shows expressions using each of the ARL operators and the corresponding Lisp forms into which they translate. This table does not cover all the possibilities but is intended as a helpful guide to expressing most rule clauses in ARL.

Predicates

parm = *value*	(SAME CNTXT *parm value*)
parm ~= *value*	(NOTSAME CNTXT *parm value*)
parm IS KNOWN	(KNOWN CNTXT *parm*)
parm IS NOTKNOWN	(NOTKNOWN CNTXT *parm*)
parm IS DEFINITE	(DEFINITE CNTXT *parm*)
parm IS NOTDEFINITE	(NOTDEFINITE CNTXT *parm value*)
parm IS DEFINITELY *value*	(DEFIS CNTXT *parm value*)
parm IS DEFINITELY NOT *value*	(DEFNOT CNTXT *parm value*)
parm IS THOUGHTNOT *value*	(THOUGHTNOT CNTXT *parm value*)
parm IS MIGHTBE *value*	(MIGHTBE CNTXT *parm value*)
parm IS VNOTKNOWN *value*	(VNOTKNOWN CNTXT *parm value*)
parm IS DEFIS *value*	(DEFIS CNTXT *parm value*)
parm IS DEFNOT *value*	(DEFNOT CNTXT *parm value*)
parm IS NOTDEFIS *value*	(NOTDEFIS CNTXT *parm value*)
parm IS NOTDEFNOT *value*	(NOTDEFNOT CNTXT *parm value*)

Figure 4-5: ARL Forms

Predicates, continued

ynparm	(SAME CNTXT *ynparm*)
~*ynparm*	(NOTSAME CNTXT *ynparm*)
ynparm IS THOUGHTNOT	(THOUGHTNOT CNTXT *ynparm*)
ynparm IS MIGHTBE	(MIGHTBE CNTXT *ynparm*)
ynparm IS VNOTKNOWN	(VNOTKNOWN CNTXT *ynparm*)
ynparm IS DEFIS	(DEFIS CNTXT *ynparm*)
ynparm IS DEFNOT	(DEFNOT CNTXT *ynparm*)
ynparm IS NOTDEFIS	(NOTDEFIS CNTXT *ynparm*)
ynparm IS NOTDEFNOT	(NOTDEFNOT CNTXT *ynparm*)
numexp1 > *numexp2*	(GREATERP* *numexp1 numexp2*)
numexp1 < *numexp2*	(LESSP* *numexp1 numexp2*)
numexp1 >= *numexp1*	(GREATEQ* *numexp1 numexp2*)
numexp1 <= *numexp1*	(LESSEQ* *numexp1 numexp2*)

[GT, LT, GE, and LE are synonyms for the above operators.]

~*predicate*	($NOT *predicate*)
pred1 AND *pred2* AND ...	($AND *pred1 pred2* ...)
pred , *pred2* , ...	($AND *pred1 pred2* ...)
pred1 OR *pred2* OR ...	($OR *pred1 pred2* ...)

Numeric Expressions

number	
numparm (as an operand)	(VAL1 CNTXT *numparm*)
numexp1 + *numexp2* + ...	(PLUS *numexp1 numexp2* ...)
numexp1 * *numexp2* * ...	(TIMES *numexp1 numexp2* ...)
numexp1 - *numexp2*	(DIFFERENCE *numexp1 numexp2*)
numexp1 / *numexp2*	(FQUOTIENT *numexp1 numexp2*)
numexp1 ↑ *numexp2*	(EXPT *numexp1 numexp2*)
- *numexp*	(MINUS *numexp*)

Actions

parm = *value* (*cf*)	(CONCLUDE CNTXT *parm value* TALLY *cf*)
parm = *value*	(CONCLUDE CNTXT *parm value* TALLY 1000)
parm ~= *value* (*cf*)	(CONCLUDE CNTXT *parm value* TALLY -*cf*)
ynparm (*cf*)	(CONCLUDE CNTXT *ynparm* YES TALLY *cf*)
~*ynparm* (*cf*)	(CONCLUDE CNTXT *ynparm* NO TALLY *cf*)
parm = *value1* (*cf1*) *value2* (*cf2*) ...	(CONCLUDE* CNTXT *parm* TALLY '((*value1 cf1*) (*value2 cf2*) ...)))
textparm = *textag* (*cf*)	(CONCLUDETEXT CNTXT *textparm* (TEXT *textag*) TALLY *cf*)
parm1 = *parm2* (*cf*)	(TRANSDIFPARM CNTXT *parm2* CNTXT *parm1 cf*)
parm = TABLE	(CONCLUDET CNTXT *switch cases* TALLY *parm values*)

Switch, cases and values filled in by an interactive routine.

Figure 4-5 continued

5. Building an EMYCIN Consultant

5.1. Organizing your Knowledge Base

A crucial first step in designing your knowledge base is *deciding what you want the system to do.* Typically it will give some sort of advice about the object that is the root context type of your system. The advice will be the value(s) of one or more "goal" parameters of this context. In this section we offer some hints about how you might want to structure the information about your consultant.

Most consultants written using EMYCIN tend to have a few key goals that they attempt to determine during the course of the consultation. These goals include the parameters encoding the kinds of advice the system is to give and the major factors that allow this advice to be inferred. In addition, these goals are, by and large, pursued in the same, fixed order for each consultation. The consultant's (implicit) *inference structure* [Bennett 78] [Bennett 80], designed by the system-builder, captures the goals of the system and their order of determination. It is a very good idea to have this inference structure well worked out before you sit down at a terminal and begin interacting with EMYCIN.

The figure below shows a schematic diagram of the inference structure for the SACON consultant. The boxes depicted with heavy borders indicate groups of data (ASKFIRST) parameters that are used to infer the major goals of the system, shown in light-bordered boxes. The possible inferences and their dependencies are indicated by the arrows. The order in which the goals are pursued is roughly top-down, left-to-right.

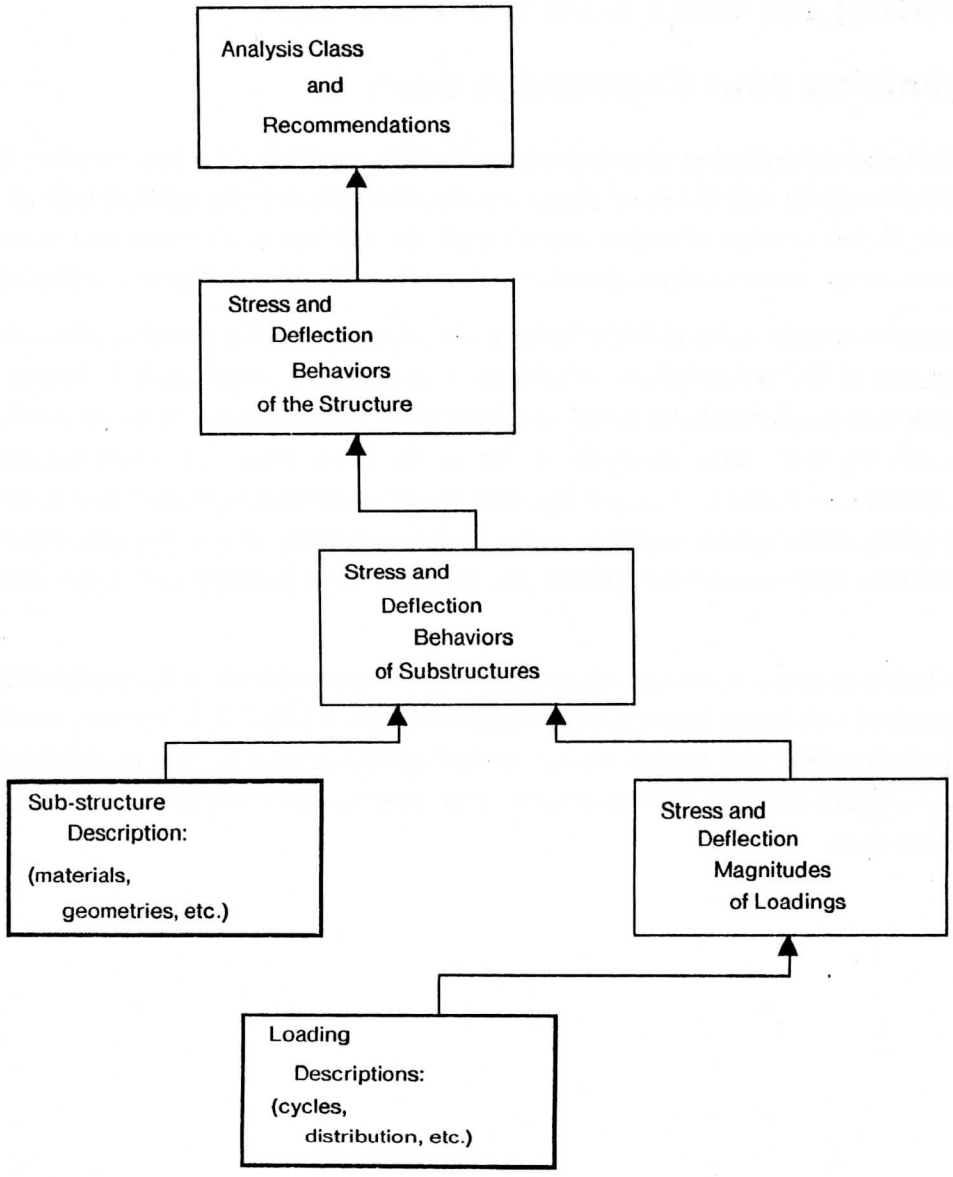

Figure 5-1: SACON Inference Structure

As mentioned in our discussion of parameters (see chapter 3), there have been several standard uses of goal parameters. These include parameters that classify data, check the consistency of both data and inferences, and determine various recommendations. Parameters have also been used to indicate that certain groups of data have been collected.

The order in which the data is collected and the goals are determined can be controlled in EMYCIN in a number of ways. The most obvious way is to indicate the data and goals using the INITIALDATA and GOALS properties, respectively, of a context type (see chapter 2). Whenever an instance of a context is created, the parameters mentioned under these properties are traced in the order indicated on the list.

An alternative method, which can yield more understandable explanations, is to encode these steps using rules. An example of one of these "goal" rules, for the SACON system, is shown below. The rule performs the final inference of analysis-class based on the stress and deflection behaviors expected in the structure as shown in the figure above. Applying this rule to a structure will cause, through the execution of other rules, the development of the tree of context instances and the determination of the stresses and deflections of the individual sub-structures.

```
RULE010

If:   1) There are no stress behavior phenomena in the structure,
      2) It is known uniquely that flexibility-changes is one of the
         deflection behavior phenomena in the structure, and
      3) Geometric is one of the types of nonlinearities in the structure
Then:   It is definite (1.0) that stiffness-degradation is one of
          the analysis-classes of the structure

PREMISE:  ($AND (NOTSAME CNTXT STRESS)
                (UNIQUE CNTXT DEFLECTION FLEXIBILITY-CHANGES)
                (SAME CNTXT NONLINEAR GEOMETRIC)
ACTION:   (CONCLUDE CNTXT ANALYSIS-CLASS STIFFNESS-DEGRADATION TALLY 1000)
```

Figure 5-2: A Goal Rule for SACON

In addition to yielding better explanations of system behavior, the use of goal rules is the only convenient way to have the consultant's behavior depend on case data. Each separate set of tasks a consultant might pursue would be represented by individual rules, conditioned with premise clauses that will "screen" a rule's application.

The rules that conclude the values of the goals and subgoals tend also to have a stereotypic structure. Each goal is concluded on the basis of a small, and usually fixed, subset of data and subgoal parameters. As the inference structure is developed, the expert should be able to tell the system-builder what the major factors are that will bear on each goal, perhaps even each possible value of the goal. Eliciting these functional groups of factors is an important step during the initial design of a new consultant.

5.2. Example: Acquiring a Business Consultant

This section presents an extended example of setting up a very simple, though contrived EMYCIN system. Its purpose is to demonstrate the typical interactions a system-builder has with the current knowledge acquisition facilities. This consultant handles a problem in the business world and employs only a single context type and a single rule. A more substantial example of the acquisition of the CLOT system is given in [vanMelle 80]. As usual, the user's input is shown in boldface and <cr> indicates the user simply typed a carriage return.

```
@<EMYCIN>EMYCIN                          [Run EMYCIN from the appropriate
                                         directory.]
EMYCIN  12-Dec-80 ...

Hi.
```

```
Do you wish to create a new knowledge base? Yes Enter a word or
phrase to describe your domain, i.e. fill in the blank in
    "this program performs a ---- consultation".  (e.g. infectious disease,
    structural analysis).
** business problem
Enter a one-word name for the root of your context tree, the central
   "object" with which the consultation is concerned (e.g. PATIENT,
   STRUCTURE):
** COMPANY                                  [The "root" of context tree]
Please enter a word or phrase as "translation" of COMPANY:
** company -- Okay? Yes                     [EMYCIN makes the obvious
                                            suggestion, we accept it. The
                                            translation might have been
                                            fancier, say "municipal agency"]
And the plural form is:
** companys -- Okay? No                     [EMYCIN doesn't know some
                                            spelling rules.]
** companies                                [So spell it right.]

Parameter group: COMPANY-PARMS -- Okay? Yes
COMPANY
PRINTID: COMPANY- -- Okay? Yes              [i.e. there will be a COMPANY-1,
                                            COMPANY-2, in consultations]
Enter a list of parameters which will always be asked of the user when a
    COMPANY is created:
COMPANY
INITIALDATA: (CNAME PROBLEM EMPLOYEES)
And now a list of parameters to infer whenever a COMPANY is created:
COMPANY
GOALS: (TOOLS)
Do you want the program to automatically display the values
    concluded for some or all of the GOALS? Yes
                                            [Automatically print the values of
                                            TOOLS at end of each consultation.]
Now please define each of the parameters listed above.  Each is a member
    of COMPANY-PARMS.
What kind of parm is CNAME -- Yes/no, Singlevalued, Multivalued, or
    Ask-All? Singlevalued
CNAME
EXPECT: ANY
CNAME
TRANS: (THE NAME OF *)
CNAME
PROMPT: (WHAT COMPANY IS HAVING A PROBLEM?)

What kind of parm is PROBLEM -- Yes/no, Singlevalued,
    Multivalued, or Ask-All? Singlevalued
PROBLEM
EXPECT: (PAYROLL INVENTORY)
PROBLEM
TRANS: (THE TYPE OF PROBLEM)
PROBLEM
PROMPT: (IS THE PROBLEM WITH PAYROLL OR INVENTORY?)
```

```
What kind of parm is EMPLOYEES -- Yes/no, Singlevalued,
    Multivalued, or Ask-All? Singlevalued
EMPLOYEES
EXPECT: POSNUMB
EMPLOYEES
UNITS: <cr>                                  [<cr> here gives the property
EMPLOYEES                                     a value of NIL]
RANGE: <cr>
EMPLOYEES
TRANS: (THE NUMBER OF EMPLOYEES OF *)
EMPLOYEES
PROMPT: (HOW MANY PEOPLE DOES * EMPLOY?)

What kind of parm is TOOLS -- Yes/no, Singlevalued, Multivalued, or
    Ask-All? Multivalued
TOOLS
LEGALVALS: TEXT                              [Values produced by CONCLUDETEXT,
                                              the results to be printed]
TOOLS
TRANS: (THE TOOLS TO USE IN SOLVING THE PROBLEM)

Okay, now back to COMPANY...                 [Now that we've defined those parms,
                                              finish defining the context type]
COMPANY
SYN: (((CNAME) (CNAME))) -- Yes, No, or Edit? Yes
                                             [The Company name (CNAME) will
                                              be used to translate a COMPANY]

Creating rule group COMPANYRULES to apply to COMPANY contexts...

..Autosave..
Please give a one-word identifier for your knowledge base files:
** BUSINESS
<EMYCIN>CHANGES.BUSINESS;1
Are there any descendants of COMPANY in the context tree ? No

Rules, Parms, Go, etc. ? Rules
Author of any new rules, if not yourself: <cr>
Will you be entering any of the rule information properties? No
                                             [this is asked upon the first
                                              entrance to the rule editor]
Rule# or NEW: NEW                            [Now enter rules to deduce each
                                              of the GOALS defined above;
                                              in this case, just TOOLS.]

RULE001
PREMISE: (PROBLEM = PAYROLL AND EMPLOYEES > 1000)
RULE001
ACTION: (TOOLS = "a large computer")

Translate, No further change, or prop name: TRANSLATE

RULE001
-------
[This rule applies to companies, and is tried in order to find out
```

```
     about the tools to use in solving the problem]

  If:  1) The type of problem is payroll, and
       2) The number of employees of the company is greater than 1000
  Then:  It is definite (1.0) that the following is one of the tools
           to use in solving the problem: a large computer

Translate, No further change, or prop name: <cr>

Rule# or NEW: <cr>                          [Finished entering rules.]

Rules, Parms, Go, etc. ? Save               [Save the knowledge base.]
<EMYCIN>CHANGES.BUSINESS;2
Rules, Parms, Go, etc. ? Go                 [Run a consultation to test the
                                             knowledge base.]
Special options (type ? for help):
** <cr>                                     [No options needed.]

20-Oct-79 14:16:48

   --------COMPANY-1--------
   1) What company is having a problem?
   ** IBM
   2) Is the problem with payroll or inventory?
   ** PAYROLL
   3) What is the number of employees of ibm?
   ** 10000000
Conclusions: the tools to use in solving the problem are as follows:
       a large computer.

Enter Debug/review, Rules, Parms, Go, etc? Parameters

Parameter name: CNAME                       [A small parameter change — we noted
Property: PROPERNOUN                         that "ibm" was not capitalized.
CNAME                                        Setting the PROPERNOUN property
PROPERNOUN: T                                will fix this problem.]
Property:<cr>

Parameter name: <cr>                        [Finished entering parameters]

Rules, Parms, Go, etc. ? Save               [Save these changes to the
                                             knowledge base.]
<EMYCIN>CHANGES.BUSINESS;3

Rules, Parms, Go, etc. ? Quit
@
        .
        .                                   [Sometime later...]
        .

@<EMYCIN>EMYCIN

EMYCIN  12-Dec-80 ...

Hi.
```

```
Should I load <EMYCIN>CHANGES.BUSINESS;3 ? Yes
File created 25-Sep-81 10:49:24
CHANGESCOMS
(<EMYCIN>CHANGES.BUSINESS;3)

Do you want to enter Rules, Parms, Go, etc (? for help) ? New consultation
 [confirm] <cr>
Special options (type ? for help):
** <cr>

23-Feb-81 10:28:37

   --------COMPANY-1--------
   1) What company is having a problem?
   ** STANFORD
   2) Is the problem with payroll or inventory
   ** INVENTORY
   3) How many people does Stanford employ?
   ** 10000

I was unable to make any conclusion about the tools to use in solving the
    problem.


Enter Debug/review phase, or other option (? for help) ? Quit
```

6. Creating and Maintaining the Knowledge Base

6.1. Putting the Knowledge Base On-line

When you feel comfortable that the major concepts and inferences have been identified, start the system and proceed first to define your context types. You will also define any parameters mentioned in each context type's INITIALDATA and GOALS lists. INITIALDATA are ASKFIRST parameters, so no further work is needed for them. For each GOAL, however, it is necessary to specify a set of rules for concluding the value of the parameter. You may be able to conclude each GOAL using only the parameters that have been defined so far. More likely, however, intermediate parameters will be needed to complete the chain of reasoning from INITIALDATA to GOALS.

Use the rule editor to define a set of rules for determining each goal. Such a set of rules can be considered as an independent "chunk" of knowledge about your domain. It is up to you to make sure that the set of rules is internally consistent and that it provides a method for deducing the parameter under all circumstances that are likely to arise.

As you define your rules, if you reference parameters you have not defined yet, you will be asked if they are indeed new parameters. If there are any, there will be a brief diversion to define the new parameter and set up its essential properties. Alternatively, you may wish to define your parameters all at once before writing rules using them, in which case invoke the parameter editor explicitly as needed. Rules and parameters may be entered in any order.

It is necessary to write a set of rules for concluding each new non-ASKFIRST intermediate parameter that you define. You may do this when each new parameter is defined, and repeat this process until you have written rules for deducing all non-ASKFIRST parameters. It will probably be easier, however, to define your knowledge base incrementally. To do this, write the rules for determining goal parameter(s) and possibly a few key subgoals. Then start testing out the knowledge base (see chapter 7) to see if the rules for your main goals are working correctly, and that the consultation interaction if proceeding as planned. If EMYCIN needs to know the value of a parameter for which you have written no rules, it will ask if it should (temporarily) prompt for the parameter. If you say yes, it gives the parameter a prompt of TEMP, which will cause a question to be asked anytime it needs the value. Later on, when you are satisfied that your system gets the right results given user-supplied values of intermediate parameters, you can then define rules for deducing these parameters (and remove the temporary PROMPT properties given them earlier.)

You also need a way to display the "results" of the consultation. The context-acquisition routine asks if you want to display the values of one or more of the GOALS you define. If you say yes, then when a consultation is run the system will display whatever values it manages to conclude for the GOALS.

There are times when you want the consultant to print conclusions only under certain conditions. The function PRINTCONCLUSIONS is provided to assist you. The system's default behavior described in the preceding paragraph is roughly the same as having an *antecedent* rule of the form:

```
PREMISE:  ($AND (ONCEKNOWN CNTXT <goal-parm1> T)
                (ONCEKNOWN CNTXT <goal-parm2> T)
                . . .
                )
ACTION:   (DO-ALL (PRINTCONCLUSIONS CNTXT <goal-parm1> T)
                  (PRINTCONCLUSIONS CNTXT <goal-parm2> T)
                  . . .
                  )
```

where <goal-parm1>, etc., are the GOALS of the root context type. This rule can serve as a model for more complex goal rules.

6.2. The EMYCIN Executive

When you start up EMYCIN you will be asked if you want to define a new knowledge base, and if so, it will invoke the routine to define the context tree. When the tree is completed, or if you answer no, you will be talking to the EMYCIN executive function. It takes simple commands to switch among the various tasks you may want to perform, e.g., enter rules, define parameters, run a consultation. As seen below, typing ? gives a list of all the possibilities. Typing L (for Lisp) will put you into the Lisp executive, in case there is something you need to do that is available only in Lisp. To return to the EMYCIN executive from Lisp type OK. In any case, when you are finished with whatever task you invoked, you will be returned to this top-level executive again to select your next task. The Typescript command will open a file which will record your interaction. The second time you give the command the typescript file will be closed.

```
Do you want to enter Rules, Parms, Go, etc (? for help): ?

    Build-system -- make a sysout
    Contexts
    Debug/review the results of the consultation
    Explanation system (QA Module)
    Go -- run the consultation program
    Important-parms -- set the result parameters for use in debugging
    Knowledge-base-lister -- output rules and parameters
    Lisp -- drop into the Interlisp executive
    Migrate-changes -- breakdown KB into separate files
    New-consultation -- like Go, but doesn't ask whether to rerun the
                        last consultation.
    OK -- Return from this EMYCIN executive
    Parameters
    Quit -- exit EMYCIN
    Rule entry
    Save changes so far on a file
    Typescript -- make (or close existing) recording of this session
    Variables, random properties, etc.
    Words -- lets you make or edit the dictionary
```

Figure 6-1: EMYCIN Executive commands

6.3. The Knowledge Base Editors

Most of the knowledge base structures in EMYCIN are entered via several "knowledge base editors", each of which have various components tailored to specific structures that are normally entered. We have tried to maintain a uniform set of conventions across these components; some are listed here.

Typically an editor will prompt for a particular component (rule number, parameter name, etc) and then prompt you for a specific command. There are basically 3 sorts of prompts issued by the program:

(1) Prompts ending in colon (e.g. "Parameter:") usually are expecting a line of input terminated by a <cr>. A <cr> on a blank line means "no value", and is *the standard way of terminating the current operation*, whatever it may be. <escape> or <altmode> may often be used to complete partially-typed words when the domain of expected responses is known (as it is for the "Parameter:" prompt, for example).

(2) Prompts ending in "?" usually are expecting just a one-letter response. Sometimes the question is simply a yes/no option (type "y" or "n"). Otherwise, the prompt asks you to select among a larger set of options. One letter in each option is capitalized to indicate the possible choices. These responses require no terminating <cr>.

(3) Prompts in brackets (e.g. "[confirm]") merely seek a confirming <cr>. Type <delete> (<rubout> on some terminals) to disconfirm. (Yes and No also work to confirm and disconfirm). These prompts indicate assumptions made by the program that you may or may not agree with and are typically inserted in order to avoid disastrous effects from typing errors or in case you wanted to do something non-standard.

Sometimes aspects of (1) and (2) are combined, as in this prompt from Variables command:

```
Add element, Set value, or propname:
```

In response, you may type A (or ADD) to select the first option, S (or SET) to select the second. Anything else will be interpreted as the name of a property whose value you want to enter or modify. Be sure to read prompts carefully before answering. If you type the new value instead of S in the example above, you will not set the value, but will be asked to define a property.

The Property Editor

Since so many structures in EMYCIN are stored on property lists, the knowledge base editors employ a central "property editor". It usually prompts by printing the name of the structure type being edited and then the property to be modified. For example,

```
RULE027
PREMISE:
```

The editor then expects either a Lisp value (a single atom, string or list), which is to be the value of the property (replacing the old value if a value already existed), or one of the following special

commands:

<cr> or OK	make no change to the property.
?	gives a brief description, if available, of the property to be entered.
P	prints the current value, if any, of the property.
E	edits the current value using the Lisp structure editor.
DEL	delete this property. Use this command rather than entering a value of NIL.
LIKE <atom>	copies the property from some other <atom>.
LIKE <prop> OF <atom>	copies from a different property name than you are currently entering.
USE <atom>, USE <prop> OF <atom>	similar to LIKE, but after copying the value it invokes the Lisp editor to edit the value.
QUOTE <atom>	used to enter <atom> as a value, rather than as a special command. For example, this is needed should you wish to enter ? as a property value.
??	prints this list of options, to refresh your memory.

Figure 6-2: Property Editor commands

Lower-case versions of these commands are also recognized. In many cases, if you just enter a new value for an existing property, you have to confirm it—this catches many typographical errors.

6.4. Saving the Knowledge Base

This section describes the contents of the knowledge-base files required for a consultant and the executive commands associated with their maintenance. Initially only a CHANGES file is required to store a knowledge base. However, as the knowledge base grows, familiarity with the other files and commands is essential.

CHANGES

During the development of a knowledge base, the EMYCIN knowledge base editors maintain a table of the structures (rules, parameters, variables, functions, etc.) that have been modified. Issuing a Save command to the executive writes a file in your login directory called CHANGES.*extension* that reflects these modifications. Loading this file (done automatically whenever EMYCIN is started) will, in conjunction with any other knowledge base files, restore your consultant to the state as of that Save command. For protection in case of system crashes, EMYCIN checkpoints your knowledge base changes, periodically saving accumulated modifications automatically.

The file name *extension* is used by EMYCIN to identify all files for this consultant: the knowledge base files themselves, listing files, hash files, typescript files, etc. It is assigned through a short interaction whenever a new knowledge base is constructed. Select a single word (for example, the name of the consultant) for identification of these files. You can change it by simply renaming all files

to use some other extension.[12]

When you first run EMYCIN several additional files are created that are used for system development. The file RULES-AVAILABLE.*extension* is created by the system to supply numbers for new rules. When you first store a case file the file CASE-GENERATOR.*extension* will be set up and it is used to assign case numbers. Both these files should reside on a central directory (MAINDIRECTORY) if there are multiple system developers to ensure that rule and case numbers do not conflict (see below).

Migration of CHANGES

As a knowledge base grows, the size of a CHANGES file can become quite large and it becomes increasingly time-consuming to both read and write this file. When this occurs and you feel that you have reached a relatively stable point of knowledge base development you should "Migrate" your knowledge base changes. Issue the Migrate command to the executive and answer the questions it asks about the disposition and types of files desired. Modifications to the various knowledge structures are then disseminated out of CHANGES and into various files as described below.

Migration performs the following functions. First, if you had specified that the core knowledge base files (RULES, PARMS, DOMAIN, and the various hash files) should be written on a central directory (by setting the variable MAINDIRECTORYusing the Variables command), EMYCIN connects to that directory. It then creates any hash files the user desires, commonly files for rule information (JUSTIFICATION, LITERATURE, etc.), the rules themselves, any dictionary files used for the QA system, and various properties of parameters and contexts. The system then writes symbolic files for RULES, PARMS, and DOMAIN. Finally the system connects to your login directory and writes a new, empty CHANGES file that will load the knowledge base from these updated files.

Migrating your knowledge base provides a convenient opportunity to create a system dictionary for use by the QA system. If you approve, the system will scan the parameters in the knowledge base, setting up pointers that the QA module can use in parsing a user question. It may occasionally stop and ask you a question such as

```
INDICATION -> INDICATE okay?
```

in its attempt to reduce words to their roots; English being what it is, especially for some of the terms in your domain, the algorithm can not be infallible. It will always ask to confirm that the reduction is correct. When finished, the dictionary resides on two hashfiles on the central directory.

Following a migration you should exit EMYCIN (using the Quit command) and start up a new version of your consultant. If you do not numerous forms of inconsistent behavior can result.

---

[12] If your system has any hash files (see below), you should change their names on the variable MYCINHASHFILES.

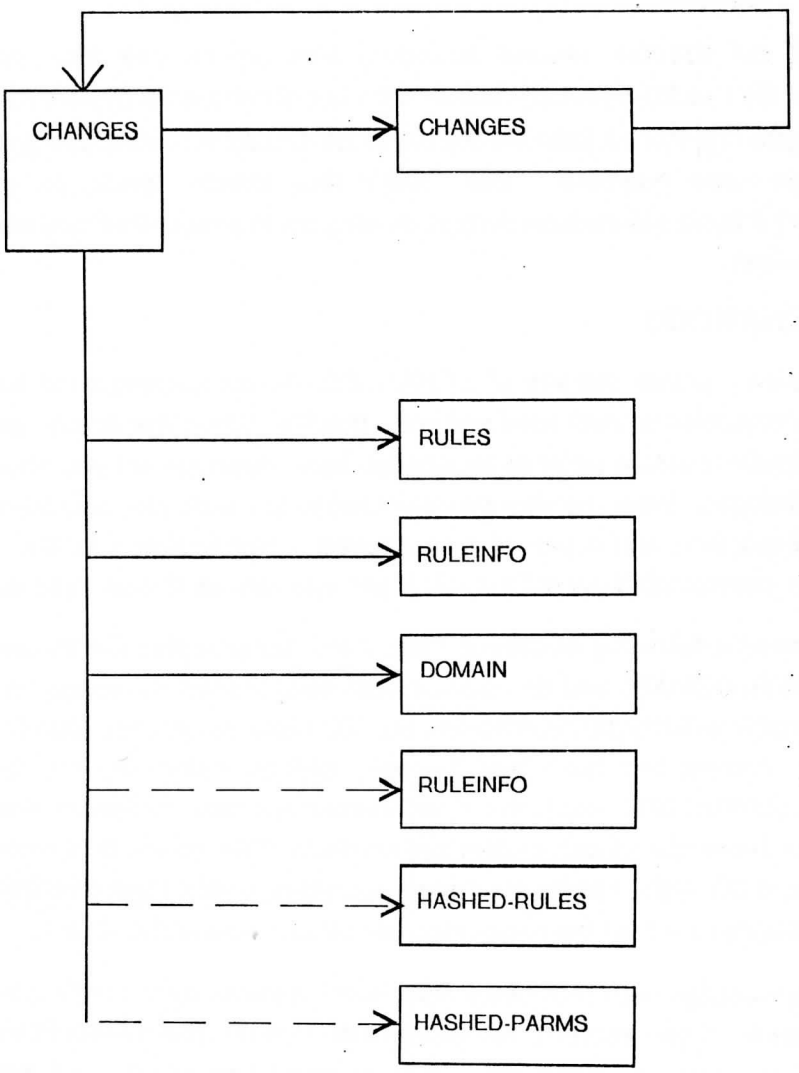

Figure 6-3: Migration of Changes

Building a consultant

If you have not elected to hash portions of your knowledge base then it will still take a substantial amount of time to load these files, although the time it takes to Save changes will be significantly reduced. You can, however, avoid this lengthy loading process without hashing you knowledge base. EMYCIN provides the capability to create an executable file that incorporates your knowledge base into a version of EMYCIN. When this file is built only CHANGES need to be loaded and written, which will contain modifications that are always with respect to that version of the knowledge base. The penalty for the reduced start up time is the large size of the executable file: on the order to 500 DEC-20 disk pages.

To construct the executable file first Migrate your changes. Exit and run a fresh copy of EMYCIN; this will reload your knowledge base. Once this has been accomplished you should immediately issue a Build command to the executive. If you have not already specified the name of your consultant (the

variable SYSTEMNAME), you will be asked to do so.[13] When EMYCIN finishes constructing the executable file you should run this file instead of EMYCIN whenever you wish to work on the knowledge base. All executive commands still work, in particular, Save and Migrate. CHANGES made against this executable file are loaded whenever you start work.

As you build your consultant it is a good idea declare those users that should have access to the EMYCIN executive. To do this, use the Variables command to set the values of the variables HACKERS and EXPERTS. Users not identified by these lists are considered GUESTS and will only be allowed to run consultations and have no access to the EMYCIN executive. The user who initially constructs the knowledge base is automatically added to the EXPERTS list.

To summarize the basic migration cycle then, modifications to a Migrated knowledge base are Saved and reloaded on CHANGES until they are once again Migrated. At this point you may elect to build a new consultant system using this Migrated system or continue to run EMYCIN, loading the knowledge base each time. To build a consultant, run EMYCIN and execute the build command after the knowledge base is loaded. Continue to make modifications to the knowledge base, remembering that every time you Migrate them you *must* rebuild your consultant.

Multiple CHANGES

In some situations there may be several experts and knowledge engineers working on a single consultant. To support system development in this setting EMYCIN allows each user to maintain a separate CHANGES file in the users login directory. The core knowledge base files (RULES, PARMS, DOMAIN, the hash files, etc.) are assumed to reside in a common directory (specified by the variable MAINDIRECTORY). The central files are written on this directory during a migration.

Note: EMYCIN does not detect or warn the user about any potential conflicts that may arise between individual CHANGES files when they are combined. It is assumed that each user is working on an independent portion of the total knowledge base and that the users themselves have coordinated their efforts through some other mechanism.

Eventually these changes will need to be combined (for example, when migrating them). This can be accomplished by having one user start up the consultant, load the knowledge base and his CHANGES. Drop into Lisp and load any other CHANGES files by executing the following expression for each user directory:

←LOAD(<*user-directory*>CHANGES.*extension*)

Return to EMYCIN using OK and continue your system development.

Note that after doing the above operations the Save command will write the *union* of all the changes loaded—not simply your own. This is typically done when migrating the system. All changes are loaded into a consultant and are then Migrated directly. Following the migration the new, empty

---

[13] Note that the file *extension* and the system name need not be the same. This is useful if you intend to maintain multiple versions of the same consultant.

version of CHANGES should be copied to all other user directories, replacing their current versions.

Hash Files

When running a consultant with a very large knowledge base it is not uncommon to exhaust working storage during a consultation. Fortunately, there are segments of the knowledge base that need not reside in working storage at all times and can be loaded on demand. These structures can be stored on hash files, which trade this space for a small amount of time required to load this information when it is to be used during a consultation. You can create the various hash files by responding appropriately to the questions asked during a migration. There is currently no convenient method for constructing and maintaining user-specific hash files beyond the standard hash files provided by EMYCIN.

The following is a list of the primary knowledge base files for a consultant, accompanied by a brief description of that file's contents. Each file is further specified by the obligatory *extension*, e.g., HASHED-RULES.*extension*.

CHANGES	The CHANGES file will accumulate the modifications you make to your (migrated) knowledge base. The files below (Rules, Parms, etc.) remain unchanged until the next migration. During migration, CHANGES is emptied and the other knowledge base files are updated to reflect the modifications that had been accumulated. Changes is updated by the Save and Migrate commands.
RULES	Unless the rules are hashed (see HASHED-RULES below), this file contains symbolic versions of the rules, the rule pointers, definitions of the rule groups, and any rule information properties. When the rules are hashed this file is used to load the rules and rule pointers from the hash files.
RULEINFO	A hash file: contains the rule information properties for all rules. Currently this includes JUSTIFICATION, LITERATURE, AUTHOR, CASES, COMMENTS, and the LASTEDIT date.
HASHED-RULES	A hash file: contains the rules themselves. Rules must be hashed to use the rule compiler.
PARMS	Unless the parameters are hashed, this file contains the definitions of all members of PARMGROUPS, CONTEXTTYPES, and AUXPARMGROUPS.
HASHED-PARMS	A hash file: contains certain properties of parameters and contexts such as PROMPT, REPROMPT, etc.
DOMAIN	This file contains numerous miscellaneous system variable definitions that help configure the consultant and modify EMYCIN for this domain. It also contains definitions of any user-defined predicate and action functions and the settings for all domain-specific knowledge structures defined by DOMAIN.VARIABLES.

Figure 6-4: Knowledge base files for a consultant

LEX-WORDS	A hash file: contains part of the dictionary used by the QA system.
LEX-CONCEPTS	A hash file: also part of the system dictionary.
RULES-AVAILABLE	Indicates what rule numbers are available when defining new rules.
CASES-GENERATOR	Indicates that case numbers are available when saving cases during a consultation.

Figure 6-4 continued

Each EMYCIN consultant should have a modified version of the file HELPFILE.HASH. It contains a variety of prompts and information for use by the help facility in EMYCIN. You may modify entries in this file using the function EDITTEXTFILE[hashfilename] in Lisp.

The Lisp function MAKELIBRARY can be used to collect a set of case files into a caselibrary hash file: *extension*-CASE-LIBRARY.<case-type> contains the cases and *extension*-SUMMARIES.<case-type> contains the case summaries. For example, MYCIN-CASE-LIBRARY.MENINGITIS and MYCIN-SUMMARIES.MENINGITIS would contain those cases used to test MYCIN on a group of meningitis cases. The function MAKELIBRARY expects the extension name for the file, the directory name where the cases are located, and a list of case numbers to build the file from. Thus, the example files would have been created by dropping into Lisp and evaluating the expression:

```
←MAKELIBRARY(MENINGITIS MYCIN <list of case numbers>)
```

6.5. Listing the Knowledge Base

The command Knowledge-base-lister is available to create readable versions of all or some of your knowledge base. This command allows you to modify several parameters that effect the appearance and content of the listings. In addition there is a standard set of defaults that you may alter. Until you have a moderately-sized knowledge base, however, the standard set of defaults should be quite adequate.

Below is a short example showing the operation of this command. In general, typing "?" at any prompt will explain the available options. Typing a carriage return at any time will select the default for option or operation.

```
Rules, Parms, Go, etc (? for help) : Knowledge-base-lister

Rules, Parameters, Both, Set-defaults, <cr>, or Abort (? for help): ?

    Your answers to the prompts below control the format of the
    knowledge base listings.  An answer of ? will give a explanation
    of the legal and default answers.  A carriage return, <cr>,
    will use the default answer for the prompt.

    Rules - to make a file with a description of the rules
    Parameters - to make a file with a description of the parameters
    Both - for both rule and parameter files
    Set-defaults - to have your following answers used to set
      the default formats of the rule and parameter listings
```

```
    Dictionary - to make a listing of the dictionary files
    <cr> - to abort


Rules, Parameters, Both, or Abort (? for help): Both

For the rules file:
Mode, Rules, Sort, File, or Linelength: Mode
one or more of E, L, T, or J: ?

    Type one or more of the following letters:
    E - English translation, also prints English
            description of the rule's SUBJECT.
    L - Lisp, prints Lisp form of the rule.
    T - Terse, prints terse (ARL) form.
    J - Justifications for rules are printed
    <cr> - Both English and Lisp.

one or more of E, L, T, or J:
Mode, Rules, Sort, File, or Linelength: Rules
Type a sequence of numbers, contexttypes, parameters, rulegroups, or a <cr>
: ?

    Typing:
    a number will have the rule with that number printed
    a contexttype causes all the rules in its RULETYPES to be printed
    a parameter will have all the rules that use the parameter in their
          premise printed,
    a rulegroup will have all the rules in that rulegroup printed
    or a <cr> to have all of the rules printed.

:       [The user specifies that all the rules are to be printed.]
Mode, Rules, Sort, File, or Linelength: Sort
NUMERIC, or SUBJECT: ?

    Numeric causes rules to be sorted by their number.

    Subject causes rules to be listed grouped by SUBJECT.
    An index links rule numbers with their subjects.

    <cr> gives the default: rules grouped by the parameter that
    triggers them.  Rules that would appear more than once this
    way are printed the first time and pointed to subsequently.

NUMERIC, or SUBJECT:  [Grouped by parameter that triggers the rule.]
Mode, Rules, Sort, File, or Linelength: Linelength
a number: ?

    The line length of the file, <cr> gives the default
    72, 58 at SUMEX, the width of the terminal if FILE=T

a number:
Mode, Rules, Sort, File, or Linelength: [the <cr> here starts the rules listing]

...
```

```
For the parameter file:
Parms, Sort, File, or Linelength: Parms
Type some parameter names or groups: ?

    To specify which parameters will be listed you may type:
    a sequence of parameter names
    a sequence of parameter groups
    or <cr>, which lists all the parameters

Type some parameter names or groups:
Parms, Sort, File, or Linelength: File
T, or a file name: ?

    T will have the parameters appear on the terminal
    any name typed will be used as a file name

T, or a file name: T
Parms, Sort, File, or Linelength: Sort
Sort the parameters by Group or alphabetically (<cr>) : ?

    Group causes parameters to be printed in their groups
          with an initial index of parameters
    <cr> has parameters listed alphabetically.

Type parameter groups wanted:
Parms, Sort, File, or Linelength: Parms
Type some parameter names or groups: FUNCTIONS
```

[we decide to list the parameters that are members of the auxiliary parameter group FUNCTIONS, i.e. new functions we have defined ourselves]

```
Parms, Sort, File, or Linelength:
May 1, 1981 2:10PM in <CSD.EMYCIN>EMYCIN.EXE.1213 by CSD.PEAIRS

                                Parameter listing
                                -----------------
FUNCTIONS
                                    FUNCTIONS
                                    ---------

CONCLUDESUBS
------------
  TRANS:  ((NUMPHRASE 5)
           every
           ((2 1))
           which is a member of ((3 1))
           is
           (COND ((MINUSP (CAR (FNTH EXPR 6)))
                  "incorrect")
                 (T "correct")))
  TEMPLATE:  (CNTXT PARM LST TALLY CF)
  CALLED-IN:  (RULE026)
  defn:  [LAMBDA (CNTXT PARM TEST-FOR TALLY CF)
                 (* edit: " 2-Oct-80 10:30")
```

```
(for PAIR in (VAL CNTXT PARM)
     when
     (MEMBER (CAR PAIR)
             (EVALV TEST-FOR))
     do
     (CONCLUDE CNTXT PARM (CAR PAIR)
               TALLY CF]
```

```
MEMBER
------
  TEMPLATE:  (ATOM LIST)
  TRANS:  (is a member of (2))
  defn:  compiled definition
```

You may discover the word SREFMARK in your rules listing. This word indicates that the rules following it are self-referencing.

7. Debugging the Knowledge Base

Once you have defined the basics of your knowledge base, you will want to debug it by running consultations, and verifying that the system makes the correct recommendations. The special options EXPLAIN, TEST, and REVIEW are designed to aid you in debugging; the question-answering (QA) facility can also be used to track down problems with your rules. Much of the time spent in debugging can be saved by running cases in BATCH mode.

7.1. Consultation Tracing

If you would prefer not to use the debugging facilities, you may be able to determine the cause of an error in a consultation's results by using the trace facility, or by looking at some of the system's dynamic data structures described in section 9.3.

There are three modes of tracing provided by the system:

TRACE 1	Prints a trace of all conclusions made.
TRACE 2	Print rule failures as well.
TRACE 3	Verbose trace of control structure's activities.

To use the tracing facility, enter TRACE followed by the desired number at any time during the consultation. The command TRACE 0 turns the tracing off. FT (for fulltrace) is accepted as an abbreviation of TRACE.[14] In addition, at any consultation question the client is asked, the user can examine the internal, dynamically-created data structures by using the EX command. You can use this command to determine the conclusions that the consultant has made about a specific context instance (e.g., EX CULTURE-6), determine the application status of a particular rule (e.g., EX RULE027), and determine what context instances a particular parameter has been traced (e.g., EX INDENT).

7.2. WHY and HOW Questions

During the consultation, EMYCIN can offer explanations of the current, past and likely future lines of reasoning. These explanations are helpful both in validating the consultant's conclusions to the client, and also to the system buider as a debugging aid. If the motivation for any question that the program asks is unclear, the user may temporarily put off answering and instead inquire why the information is needed. Since each question is asked in an attempt to evaluate some rule, a first approximation at an answer is simply to display the rule currently under consideration. The program can also display what reasoning led to the current point, and what use might later be made of the information being requested.

The following example from MYCIN shows the WHY command being used to ascend the current reasoning chain. Each successive WHY asks why the subgoal of the previous response is being sought; the answer is, of course, the rule whose premise needs the subgoal. Eventually, WHY

---

[14] Do not confuse this TRACE option with the mechanism of tracing (finding out) a parameter's value.

reaches the top of the subgoal tree, i.e., the main goal of the consultation. The user can also ask HOW any of the clauses mentioned in the rules displayed by WHY is known or will be found out. The two commands together allow the user to step through any path in the system's reasoning that is "reachable" from the current state.

```
    34) Was penicillinase added to this blood culture (CULTURE-1)?
    ** WHY

[i.e. WHY is it important to determine whether penicillinase was
      added to CULTURE-1?]
[3.0] This will aid in determining whether ORGANISM-1 is a
      contaminant.
  It has already been established that
     [3.1] the site of CULTURE-1 is blood, and
     [3.2] the gram stain of ORGANISM-1 is grampos
  Therefore, if
     [3.3] penicillinase was added to this blood culture
  then
     there is weakly suggestive evidence (.4) that ORGANISM-1 is a
               contaminant
[RULE039]

    ** WHY

[i.e. WHY is it important to determine whether ORGANISM-1 is a
      contaminant?]

[4.0] ... in order to determine whether there is significant disease
associated with this occurrence of ORGANISM-1.
  It has already been established that
     [4.1] this blood culture was taken from a sterile source
  Therefore, if
     [4.2] ORGANISM-1 is not a contaminant
  then
     there is strongly suggestive evidence (.9) that there is
     significant disease associated with this occurrence of ORGANISM-1
[RULE032]

    ** WHY

[i.e. WHY is it important to determine whether there is significant
     disease associated with this occurrence of ORGANISM-1?]

[5.0] ... in order to determine the organisms which require therapy.
  It has already been established that
     [5.1] the infection is known, and
     [5.2] the identity of ORGANISM-1 is known
  Therefore, if
     [5.3] there is significant disease associated with this
               occurrence of ORGANISM-1
  then
     ORGANISM-1 is one of the organisms which require therapy
[RULE090]
```

```
   ** HOW 5.2
[i.e. HOW was it established that the identity of ORGANISM-1 is
      known?]

Question 9 was asked, but you weren't sure; so the following was used:

  [5.5] RULE084 was used to conclude that the identity of ORGANISM-1
                is Streptococcus-group-a (.29).
```

7.3. The EXPLAIN, TEST, and REVIEW Options

These special options are used for providing information about the values of parameters at the end of a consultation. The EXPLAIN option provides terse explanations about how the parameters' current values were concluded. With the TEST option specified, the system compares the current value of each parameter with the stored value for the case. REVIEW provides explanations, and if these are incorrect, helps the expert to locate the problems in the knowledge base.

You can tell the system which parameters you consider to be the "results" of a consultation by using the Important-parms command. Using it, you will define IMPORTANTPARMS, a variable that will specify to the EXPLAIN, TEST and REVIEW special options which parameters to examine. If you use any of the special options without defining IMPORTANTPARMS, you will be asked which parameters are of interest. If your system has a dictionary, you may enter this information in English. Otherwise you will have to use the Lisp names of all parameters.

Defining the consultant's IMPORTANTPARMS

Use the Important-parms command from the EMYCIN executive to the result parameters for your consultant. You may specify numerous (even overlapping) groups of parameters for different options and screened by different conditions. This command will prompt you these groups and, for each group, request the following information.

1. which special options (EXPLAIN, TEST, REVIEW, or ALL of them) this group of parameters will apply to,
2. the parameters to be EXPLAIN'ed, TEST'ed, or REVIEW'ed,
3. the conditions under which to execute the special option(s), and
4. the context types to display the parameters from.

You may specify a condition that must be true in order for the option to have effect on this group. Most of the time you will always want the option to occur; you may indicate this by using the value T. You may also write an arbitrary Lisp expression that will be evaluated and, if non-NIL, will allow the option to occur. This expression may reference the atom CNTXT, which is bound to the context instance to be tested before the condition is evaluated. The function SAMEP can be called to test the value of a parameter. It is true if $AND would consider the analogous SAME clause true (remember that SAME returns a number, so cannot be used as a predicate outside of $AND).

An example from MYCIN

```
Rules, Parms, Go, etc. : Important-parms
Show, Add, Delete, or Change: Show
1)  Always EXPLAIN TEST
       the organisms that will be considered to be causing the infection for
       the purposes of therapy selection
2)  Whenever (SAMEP CNTXT (QUOTE REQTHER)) EXPLAIN TEST
       the organisms (other than those seen on cultures or smears) which
       might be causing the infection, and
       the type of the infection
    for every instance of
       positive cultures obtained from it,
       pending cultures of it, and
       suspected infections without microbiological evidence.
3)  Whenever (SAMEP CNTXT (QUOTE REQTHER)) EXPLAIN TEST
       the identity of the organism
    for every instance of
       organisms isolated from it, and
       organisms noted on smears of it.
4)  Whenever (OR (GETOFFSPRING CNTXT (QUOTE KNOWNORG))
                 (GETOFFSPRING CNTXT (QUOTE SMEARORG))) REVIEW
       the organisms which require therapy
5)  Always REVIEW
       the organisms isolated from the culture should be considered for
       therapy
    for every instance of
       positive cultures obtained from it, and
       pending cultures of it.
6)  Whenever (SAMEP CNTXT (QUOTE REQTHER)) REVIEW
       the organisms (other than those seen on cultures or smears) which
       might be causing the infection
    for every instance of
       positive cultures obtained from it, and
       pending cultures of it.
7)  Always REVIEW
       the organisms (other than those seen on cultures or smears) which
       might be causing the infection
```

The EXPLAIN Option

The EXPLAIN debugging option may be selected by including EXPLAIN among the special options at the beginning of the consultation. At the end of the consultation session and before entering QA, EXPLAIN provides terse explanations telling how the system reached its conclusions about the results of the consultation. In most cases, the information provided by EXPLAIN should be sufficient for the system builder to tell whether the system performed correctly, and to pinpoint the source of any errors that may exist. Each rule contributing to the conclusion is printed along with the cumulative certainty factor after the rule succeeded, the last question asked before the conclusion was made, and the value of TALLY when the rule succeeded.

This option is intended for the experts building the consultation system who are familiar with the internal parameter names. For fuller explanations, the TEST option may be selected instead of

EXPLAIN.

Below is an example of the type of explanations that are provided by the EXPLAIN option in the MYCIN system.

```
EXPLAIN the results of the consultation.  Would you like to know
how the values were determined?  Options are Y (yes, all items),
N (no items), and A (ask for each item).
** YES

COVERFOR of CULTURE-1
PSEUDOMONAS-AERUGINOSA (.74)
                RULE557 (.49) [Q 49] (Tally:  491)
                RULE578 (.74) [Q 52] (Tally:  491)
DIPLOCOCCUS-PNEUMONIAE (.74)
                RULE507 (.74) [Q 41] (Tally:  737)
E.COLI (.74)
                RULE557 (.74) [Q 49] (Tally:  737)
KLEBSIELLA-PNEUMONIAE (.49)
                RULE557 (.49) [Q 49] (Tally:  491)
```

This tells us for example that for PSEUDOMONAS-AERUGINOSA the final CF is .74. The answer to question 49 had no effect on the CF for PSEUDOMONAS-AERUGINOSA. Before rule 557 was used the Tally was 491, afterwards the certainty factor was .49. But the answer to question 52 allowed rule 578 to raise the Tally from 491 to a CF .74.

The TEST Option

The system builder can use the TEST option to compare the results of the current run with the stored correct results and receive an explanation for any discrepancies between the two. If differences exist, the appropriate QA functions are called to explain why new values were concluded or why correct values were not concluded. The correct results of a case are stored by the REVIEW debugging option; if the correct results have not been stored, TEST simply explains how the system's conclusions were made. TEST also reports whether there is any change in the results since the last time a consultation was run on the same case. After these explanations, the system summarizes the problems that exist and gives the user the option of listing the pertinent rules. The user is also given the option of trying to fix the problem using the same mechanism that is used by the REVIEW option.

Once the system is able to reach the correct results for a case, then, TEST can be used to verify that subsequent changes to the knowledge base do not degrade the systems performance on this case.

Below is an illustration of the type of explanations that are provided by the TEST option. This explanation was generated by the second entry on MYCIN's IMPORTANTPARMS (see above).

```
Testing the parameter:
     "the organisms (other than those seen on cultures or smears) which
      might be causing the infection".
In the context instances of positive cultures.
Whenever (SAMEP CNTXT 'REQTHER)).
That is whenever the parameter
```

```
  "the organisms isolated from the culture should be considered for therapy"
   has a cf greater than .2


I concluded that the identity of ORGANISM-1 is:  E.coli (.43), Pseudomonas-
     aeruginosa (.32), Klebsiella-pneumoniae (.24).
[No change.]
In a previous debugging session, I was told that I should have concluded
     that the identity of ORGANISM-1 is:  Klebsiella-pneumoniae,
     Pseudomonas-aeruginosa, and Bacteroides-fragilis; also, the identity of
     ORGANISM-1 is not any of the following:  Hemophilus-influenzae, E.coli.
[missing:  Bacteroides-fragilis]
[spurious:  E.coli]
I will attempt to explain why I made the wrong conclusion.
The following rules could have concluded about:
     the likelihood that the identity of ORGANISM-1 is Bacteroides-fragilis
Rules:  209, 208, 164.  However, none of these succeeded in the context of
     the pending csf culture (CULTURE-1).  If you would like an explanation
     for why any of these rules failed, please enter their numbers:
      ** NONE

The following rules concluded about:
     the likelihood that the identity of ORGANISM-1 is E.coli

RULE345 was used to conclude that the identity of ORGANISM-1 is E.coli
     (.24).  The last question asked before the conclusion was made was 46.
RULE357 was used to conclude that the identity of ORGANISM-1 is E.coli
     (.43).  The last question asked before the conclusion was made was 48.

The parameters most frequently used in these rules were:
     the infection which requires therapy
     the stain of the organism
     the morphology of the organism

Would you like a summary of ALL the parameters used by these rules?   ** NO

I concluded that the type of the infection is bacterial (.98), viral (-.76),
     fungal (-.83) and tb (-.83).
[No change.]
In a previous debugging session, I was told that I should have concluded
     that the type of the infection is bacterial; also, the type of the
     infection is not any of the following: tb, fungal, viral.
[Current conclusion is correct.]

Problems still exist with the rules for deducing
     the identity of the organism
Would you like a listing of the rule set for concluding this parameter?
** NO
Would you like to try to correct the problems now?
** NO
```

The REVIEW Option

This option allows the user to review the system's conclusions about the results of a case. Each conclusion is presented, and the user is asked to judge whether it is correct. If it is not, the user supplies the correct result, and is given the choice to fix it without system assistance, let the system guide the debugging, or defer its correction. A description of each deferred problem is sent to the message file UNSOLVED-PROBLEMS.extension on the connected directory. The correct results of the case will be saved on the case file for use in future debugging sessions (this information is used by both the REVIEW and the TEST options).

After all results have been reviewed, if problems existed and changes were made that might fix them, the consultation is re-run and the new results are reviewed. This process iterates until all problems are fixed, or the user asks to stop. If unsolved problems remain at the end of the debugging session, the system will summarize them and give the user the option of listing the pertinent rule sets.

7.4. The QA Module

EMYCIN systems that have dictionaries can use the question-answering (QA) system.

The QA Module accepts simple English-language questions (a) dealing with any conclusion drawn during the consultation, or (b) about the domain in general. The level of explanation is again that of the rules, presumably comprehensible to anyone familiar with the domain, even if not familiar with the intricacies of the EMYCIN system. The questions are parsed by pattern matching and keyword lookup, using the dictionary that defines the vocabulary of the domain.

Examples of general questions about the MYCIN and SACON domains are shown below.

```
** IS BACTEROIDES AEROBIC?

No, the aerobicity of Bacteroides is anaerobic.

** WHAT ORGANISMS ARE LIKELY TO BE FOUND IN THE THROAT?
The likely pathogens associated with throat are:  DIPLOCOCCUS-
     PNEUMONIAE, STREPTOCOCCUS-GROUP-A, NEISSERIA-MENINGITIDIS.


** HOW DO YOU USE THE GEOMETRY OF A SUB-STRUCTURE?
There are 5 rules which use
     the geometry of the sub-structure

53, 48, 102, 101, 100

The parameters concluded by these rules are:
     the shape of the sub-structure (53 45)
     the analysis recommendations to be considered when preparing
          the structure for modelling (102 101 100)

Which of the rules do you wish to see?
      ** 53

RULE053
```

```
If:  1) Buckling is one of the deflection phenomena in the
        structure, and
     2) You have examined the sub-structures, and have found one
        for which the geometry of this sub-structure is curved
Then: The following is one of the analysis recommendations to be
        considered when preparing the structure for modelling:
        Buckling estimates should be based on incremental load
        analysis.
```

Examples of questions about a particular MYCIN consultation are shown below.

**** IS ORGANISM-1 A STREPTOCOCCUS?**

```
No.  The identity of ORGANISM-1 is E.coli (.74) Bacteroides (.7)
    Klebsiella-pneumoniae (.4) Pseudomonas-aeruginosa (.22)
    Enterobacter (.21) Proteus-non-mirabilis (.19) Serratia (.03)
    Salmonella (.02)
```

**** WHY DID YOU RULE OUT STREPTOCOCCUS AS A POSSIBILITY FOR ORGANISM-1?**

```
RULE033 could have been used to determine that the identity of
    ORGANISM-1 is Streptococcus.  However, clause 2 of rule033 ["the
    morphology of the organism is coccus"] was already known to be
    false for ORGANISM-1, so the rule was never tried.
```

The Special Options available in the QA module

```
ADVICE - prints suggestions for using the QA system effectively
QT - prints examples of question types recognized by the system
EQ <numbers> - prints an explanation of the specified
          question(s) from the consultation, e.g., EQ 8 13
LP <object type> - prints list of attributes associated
          with object type, e.g. LP CULTURE.  (Object types in MYCIN
          are patient, infection, culture, organism, and drug.)
PR <numbers> - prints specified rule(s), e.g., PR 373 49 87 2
JUSTIFICATION - when rules are printed out, include their
    justifications
LITERATURE - when rules are printed out, include their literature
    references
NOJUST - stop giving justification and/or literature with rules
TERSE - print rules in terse format
VERBOSE - turns off terse printing of rules
STOP - escape from QA-module
EXPLAIN - print a terse  explanation of how the results of the
    consultation were concluded
TEST - test the results of the consultation by comparing them with
    stored correct results for this case and explaining differences
REVIEW - review the results of the consultation.  The system will
    present its conclusions and you will be allowed to say whether these
    are correct.  If not, you will be allowed to try to fix the error.
HELP - prints this list
```

Figure 7-1: QA Options

Legal forms for the QA system

Outlined below are the types of questions about a consultation that can be recognized and answered. A few examples of each type are given. All questions about the consultation should mention a particular object (organism, culture, etc.) from the consultation. Furthermore, the question must indicate the parameter or rule of interest. (The LP option can be used to find out which parameters the different types of objects may have, e.g., LP ORGANISM). In the examples below, <object> indicates some object which was discussed in the consultation; <parameter> is some clinical parameter of this object; <rule> is one of the system's decision rules.

1) **WHAT IS** <parameter> **OF** <object>
```
WHAT IS THE GENUS OF ORGANISM-1?
IS ORGANISM-1 CORYNEBACTERIUM-NON-DIPHTHERIAE?
```

2) **HOW DO YOU KNOW THE VALUE OF** <parameter> **OF** <object>
```
DID YOU CONSIDER BACTEROIDES AS A POSSIBILITY FOR ORGANISM-1?
WHY DON'T YOU THINK THAT THE SITE OF CULTURE-1 IS URINE?
WHY DID YOU RULE OUT STREPTOCOCCUS AS A POSSIBILITY FOR
   ORGANISM-1?
```

3) **HOW DID YOU USE** <parameter> **OF** <object>
```
DID YOU CONSIDER THE FACT THAT PATIENT-1 IS A COMPROMISED
  HOST?
HOW DID YOU USE THE AEROBICITY OF ORGANISM-1?
```

4) **WHY DIDN'T YOU FIND OUT ABOUT** <parameter> **OF** <object>
```
DID YOU FIND OUT ABOUT THE PATIENT'S CBC?
WHY DIDN'T YOU NEED TO KNOW WHETHER ORGANISM-1 IS A
   CONTAMINANT?
```

5) **WHAT DID** <rule> **TELL YOU ABOUT** <object>
```
HOW WAS RULE 178 HELPFUL WHEN YOU WERE CONSIDERING ORGANISM-1?
DID RULE 116 TELL YOU ANYTHING ABOUT INFECTION-1?
WHY DIDN'T YOU USE RULE 189 FOR ORGANISM-2?
```

Questions which do not refer to an object from the consultation are assumed to be general questions about MYCIN's knowledge base. Examples of such questions are listed below.

```
IS BLOOD A STERILE SITE?
WHAT ARE THE NONSTERILE SITES?
WHAT ORGANISMS ARE LIKELY TO BE FOUND IN THE THROAT?
IS BACTEROIDES AEROBIC?
WHAT METHODS OF COLLECTING SPUTUM CULTURES DO YOU
   CONSIDER?
HOW DO YOU DECIDE THAT AN ORGANISM MIGHT BE STREPTOCOCCUS?
WHY DO YOU ASK WHETHER THE PATIENT HAS A FEVER OF UNKNOWN
   ORIGIN?
DO YOU USE GRAM STAIN TO DETERMINE GENUS?
WHAT IS THE RECOMMENDED DOSEAGE OF GENTAMICIN?
WHAT DRUGS WOULD YOU CONSIDER TO TREAT E.COLI?
```

Figure 7-2: Legal question forms for QA

Using the QA system Effectively

1. Examine the system's reasoning by asking how the system *concluded* values for the parameters or how parameters were *used*. Remember to mention the object for the parameter (e.g., "How did you use the identity of ORGANISM-1" not "How did you use identity").
2. Use the LP option to become familiar with the parameters that are associated with each type of object (drug, infection, etc.). For example, LP ORGANISM will list the parameters which an organism may have.
3. Use the QT option to see what types of questions the system can recognize and answer. The examples it gives will illustrate how questions can be asked to trace the system's reasoning.

7.5. Debugging with Batch

The main purpose of the batch facility is for debugging the knowledge base. It allows you to test the affect of knowledge base changes on stored cases without having supervise the re-running of the cases. The REVIEW debugging option may be used to record on each case file the "correct" results of a consultation on that case. The TEST option is automatically selected in batch runs to compare results of the current run with the stored correct results and to explain any discrepancies between the two. If the correct results have not been stored, the system simply explains how its conclusions were made (the EXPLAIN option provides this same information in a more concise format, and can be used in batch runs instead of TEST, see section 7.3).

Once the system is able to reach the correct results for a case, batch runs can be used to verify that subsequent changes to the knowledge base do not degrade the system's performance on this case. When changes do occur in the results, the TEST option provides explanations of why each change occurred. This helps the expert locate problems in the knowledge base.

All new questions during the consultation are answered UNKNOWN (and the questions are noted). The QA module is not called, but EXPLAIN is called if EXPLAIN is one of SPECIAL.OPTIONS. The output of the consultation is directed to a file BATCH.nnn, where nnn is the number of the stored case. A brief message summarizing the results is sent to a message file BATCH.REPORTS on the connected directory (this file will be created if it does not exist). The message includes new questions which were asked, the "results" of the consultation as specified by IMPORTANTPARMS and PDTCHANGEPROPS, and an explanation of differences between the correct results that are stored on the case file and results of this consultation.

BATCH *must* be invoked underneath the "Special options" prompt. The special option BATCH calls BATCH, passing the remainder of the command line as the NUMBERS argument. Fancier invocation is possible by doing a ↑Y at that point and calling BATCH from Lisp. A typical special options line might be:

```
** TRACE 2 SUMMARY BATCH 201 202 203
```

Running a Batch Job

If you are unfamiliar with using the batch system, consult your operating system's documentation. Define a BATCH.CTL file as described below, then run the job by typing (at SUMEX):

```
@SUBMIT                          [@ and * are prompt characters]
*ctl BATCH.CTL
*after 3:00                      [or some other !!LATE!! hour]
*
```

On a TOPS-20 system:

```
@submit batch /after 3:00 /time 00:60:00
```

To create a control file (BATCH.CTL), use any text editor. The control characters "↑Y" and "↑W" in the files can be entered as "↑" followed by "Y" and "W" respectively. They are interpreted as control characters when the batch job is running. A line that should be followed by return is preceded by *. Below is a simple control file for a batch run.

```
@<EMYCIN>EMYCIN
*Go
*BATCH 104
```

The BATCH function itself, which performs all the work, expects the following arguments:

BATCH[NUMBERS,HACKERFLG,CHANGESONLY,DIRECTORY,NOEXPL]

NUMBERS — is a list of numbers of cases to run. It may be NIL, in which case the numbers are taken from a file BATCH.NUMBERS, which is simply a list (enclosed in parens) of numbers; as each case is run, its number is crossed off the file, so that the next time you run BATCH, you won't get the same numbers. When the file is exhausted, it is deleted.

HACKERFLG — is usually NIL for BATCH runs. Set this to T if you are running BATCH "interactively".

CHANGESONLY — means to keep the output file BATCH.nnn only if something has changed since this case was stored.

DIRECTORY — may be used to specify a directory other than the connected directory where the case files are to be found.

NOEXPL — may be set to T if you don't want the system to use the TEST option to be used.

If changes occur in the results, an updated case file is written. If you don't like this feature, you can disable it by doing

←MOVD(NILL DUMPCASEFILE)

before calling BATCH.

BATCH stops running cases (i.e. it will quit after the current case finishes) if a ↑Z interrupt is received. At SUMEX it is possible to tell the TENEX batch system to send a ↑Z when time is nearly up. BATCH will also stop if a consultation is aborted for any reason (a typical case is STORAGE FULL, and this usually happens before time runs out (default time in batch on Sumex is 1 hour of cpu time)).

Using the Batch Explanation Facility

You can tell the system which parameters you consider to be the "results" of a consultation by giving values to the atoms IMPORTANTPARMS and PDTCHANGEPROPS.

If your system has specialized functions that create lists (other than the value of some parameter) that are considered important results, PDTCHANGEPROPS can be used. Most EMYCIN systems do not have such lists. If you would like to use this feature set or edit the variable PDTCHANGEPROPS using the Variables command. The format of the variable is as follows:

PDTCHANGEPROPS is a list of elements, each of which corresponds to some result from the consultation. An element has the form (<newvalue> <storedvalue> . <explfn>).

<newvalue>	is an atom whose value at the end of the consultation will be a "result" list
<storedvalue>	is an atom whose value is that same "result" list as stored on the case file
<explfn>	is optional, and is a function that should be called from BATCH to explain why <newvalue> and <storedvalue> are not the same.

MYCIN's PDTCHANGEPROPS is:

```
((NEWITEMLIST ITEMLIST . ITEMLISTCHANGED)
 (NEWREC REC . RECCHANGED))
```

Note that EMYCIN system builders must create functions that set the value of each <newvalue> during a consultation. Functions already exist for moving this value to <storedvalue> before a case file is created; this is done automatically. BATCH automatically compares <newvalue> and <storedvalue> and report differences in the values. System builders who want some explanation of why the differences occurred must write their own <explainfn>s.

If you have case files that were created before you defined IMPORTANTPARMS and PDTCHANGEPROPS, they have to be updated to include the results. The easiest way to do this is to rerun them all in BATCH with NOEXPL set to T.

A more complicated BATCH control file is shown below. The line numbers are just included to facilitate discussion; do not type them in your file. Lines that are input to Lisp which do not have to be followed by a carriage return (because they contain balanced parens, for example) are preceded by "=". You may insert lines to execute any Lisp commands before calling BATCH, for instance, loading files (Lines 5-6). You may enable ↑Z as discussed earlier (Line 2).

If BATCH.NUMBERS is a very long list and your system is large, you may run out of Lisp space before running all the cases. In order to restart the process after such an abort, add the Line 8, and repeat all the commands for running the job (Lines 10-16).

```
1       @<EMYCIN>EMYCIN
2       %DEFAULT Z
3       *Go
4       =↑Y
5       =(LOADEM TEMP)
6       =(SETQQ SPECIAL.OPTIONS (EXPLAIN))
```

```
7       =(BATCH NIL NIL NIL 'MYCIN T)
8       =(OR (INFILEP 'BATCH.NUMBERS T) (JSYS 3 -1]
9
10      @<EMYCIN>EMYCIN
11      %DEFAULT Z
12      *Go
13      =↑Y
14      =(LOADEM TEMP)
15      =(SETQQ SPECIAL.OPTIONS (EXPLAIN))
16      =(BATCH NIL NIL NIL 'MYCIN T)
```

8. Consultation Options

8.1. Consultation Special Options

Before running a consultation a variety of special options may be selected as shown below. Special options may be used to select a case to rule (e.g., NEW, OLD), to modify the manner of interaction with the consultant (e.g., NOPR, TAB), and to select a specific facility to run following a consultation (e.g., REVIEW, QA). The effect of these commands are described in following sections.

```
Rules, Parms, Go, etc. ? Go [confirm]
Special options (type ? for help):
** ?
For a regular business problem consultation, just type <cr>.

You may request special features by entering one or more of the
  following options (separate options with spaces, end with carriage
  return):

 BATCH -   run case in batch mode, should be followed with case numbers
 CRT -     the system you are using a display terminal,
 EXPLAIN - after the consultation, print a terse explanation
           of how the results were concluded
 I -       requests instructions be printed
 KEEP -    causes old cases to be remembered after they are run,
           allowing them to be rerun on the next consultation
 n -       (a number) reconsider previously-saved case n
 NOFIX -   like OLD, but will not ask if changes should be made to the case
 NOPR -    do not print out old questions and answers when running old cases
 OLD -     consider a previously-saved case (number(s) will be requested)
 QA -      enter the question/answer module immediately, skipping the
           consultation
 REVIEW -  after the consultation, review the results.  The system
           will present its conclusions and you will be allowed to
           say whether these are correct.  If not, you will be
           allowed to try to fix the error.
 REVISE -  allows answers from old cases that are now illegal to be accepted
 SAVE -    create and save file(s) for cases discussed in this consultation
 SUMMARY - summarize old patient data, rather than printing out
           each question and answer
 TAB -     tabular entry mode
 TER -     enter terse mode
 TEST -    after the consultation, test the results by comparing them with
           stored correct results for this case and explaining differences
 TRACE # - display the conclusions that the system makes,
           must be followed by a number from 0 to 3
 TS -      write out a typescript file of consultation
 UPDATE -  update old patient with new information
```

Figure 8-1: Special Options Available Before a Consultation

```
Rules, Parms, Go, etc. ? Go [confirm]
Special options (type ? for help):
** I
```

Please answer the following questions, terminating each response with RETURN. To correct typing errors, use the DELETE key to delete single characters, <ctrl>W to delete a word, and <ctrl>Q to delete the whole line.

If you are not certain of your answer, you may modify the response by inserting a certainty factor (a number between 0 and 1.0) in parentheses after your response. Absolute certainty (1.0) is assumed for every unmodified answer. It is likely that some of the following questions can not be answered with certainty.

You may change an answer to a previous question in two ways. If the program is waiting for a response from you (that is, has typed "**"), enter CHANGE followed by the number(s) of the question(s) whose answers will be altered. You may also change a previous answer at any time (even when the program is not waiting for a response from you) by typing <ctrl>F (Fix), which will cause the program to interrupt its computation and ask what you want to change. (If the response to <ctrl>F is not immediate, try typing the RETURN key in addition.) Try to avoid going back because the process requires reconsidering the case from the beginning and therefore may be slow.

Note that you may also enter UNK (for UNKown) if you do not know the answer to a question, ? if you wish to see a more precise definition of the question or some examples of recognized responses, ?? if you want to see all recognized responses, the word RULE if you would like to see the decision rule which has generated the question being asked, the word WHY if you would like to see a more detailed explanation of the question, or the letters QA if you would like to interrupt the consultation in order to ask questions regarding the decisions made so far in the consultation. If you are ever puzzled about what options are available to you during a consultation, enter the word HELP and a list of options will be listed for you.

```
Sample Response [user input follows the "**"]
  Does the patient have a risk factor for tuberculosis?
  ** ?
  One or more of the following are considered risk factors for tb:
  a) positive PPD (5TU), b) history of close contact with a
  person having active tb, c) household member with a past
  history of active tb, d) chest X-ray showing apical scarring,
  e) granulomas seen on biopsy of any organ tissue.

  Expected responses are: YES NO
  Enter HELP for user options.
  ** YES
```

Figure 8-2: Instructions for Running Consultations

```
23-Feb-81 14:41:26

   --------COMPANY-2--------
   1) What company is having a problem?
   ** HELP

 SUMMARY:
   (type ctrl-O to abort printout)
   ?       - Rephrases the question and gives examples of recognized
             responses
   ??      - prints a list of all recognized responses
   CHANGE # - go back and re-request answer to question number #
   DEBUG   - return to the EMYCIN executive, later use of the OK
             command in the executive will return you to this
             consultation.
   EX <name> - examine the properties of <name>; a parameter, rule, or
               context instance.
   EXPLAIN - provides a more detailed explanation of a previous
             answer given by a WHY command.
   FORGET - resets the explanation of the reasoning chain back to the
            lowest level, as if you never used the WHY/EXPLAIN commands.
   HELP    - prints this list.
   HOW #   - explains HOW the system will achieve a goal referred to
             by number # in a previous explanation.
   QA      - program enters question-answering mode.
   RESTART - end the consultation and start a new one.
   REVIEW - review the system's conclusions about this case.  You will be
            asked to specify which conclusions are of interest, and then to
            judge whether they are correct.  If not, you will be allowed to
            try to fix the error.
   RULE    - prints the current decision rule.
   SAVE    - Halts the program, saving the current case on a disk
             file, retrievable at a later date.
   STOP    - Halts the program without completing the consultation.
   TAB     - tabular entry mode.
   TRACE - have rule conclusions displayed, must be followed by a
           number from 0 to 3.
   WHY   - gives high-level explanation of the current reasoning
           chain that provoked this question.
   UNK   - answer not known

   **
```

Figure 8-3: Possible Answers During a Consultation

8.2. Maintaining Consultant Cases

The SAVE Option

The Save option allows you to store consulations so they may be rerun later. You may give SAVE as a special option before running the consultation. First you are asked for a description of the consulation that is about to occur. The whole consulation that then transpires is recorded and stored on a file. In the future, you need only type the case number to the "Special Options:" prompt to have the case rerun. The description you entered will be reprinted before running the stored case.

You may also say SAVE as an answer to a question during a consultation. In this case, only the answers to questions already given during the consultation will be stored.

The UPDATE Option

The UPDATE option allows the user to update stored case files. It is selected by entering UPDATE and the number of the case file as special options at the beginning of the consultation. In UPDATE mode, the system first summarizes information that is stored on the case file and interacts with the user to update this information. After this phase, the consultation is run on the updated case using whatever other options were specified (eg SUMMARY, TER, TAB, NOPR, etc.).

In the UPDATE phase, output is similar to a consultation run in SUMMARY mode. All information on the case file is printed, one context at a time. If a CNTXTSUMMARY property exists for a context, then it is used to print an "organized" summary of the parameters (see section 8.3 for an explanation of the CNTXTSUMMARY property). If there is no such property, the order in which parameters of a context are printed is arbitrary. If the answer stored for a parameter is UNKNOWN or not certain (CF < 1.0), or if the parameter has a non-NIL UPDATE property, the user is asked if the value should be updated. After information has been summarized for each of the contexts, the user is given the option of changing or deleting any parameter. The system prompts for one parameter at a time. Systems that have dictionaries can use English descriptions of the parameters or the Lisp name. If there is no dictionary (or if the dictionary is not available), only Lisp parameter names are accepted.

A new "time of the consultation" is stored if the updated case is saved. Note that parameters whose values are dates are automatically updated—the values are stored as "number of days ago" and this number is calculated relative to the new time of the consultation. If your system has parameters whose values are "number of days ago" and which are entered as number of days ago rather than as dates, these parameters should be given the UPDATE property. Any parameter that is likely to change over time should also be given the UPDATE property.

It is possible to save incomplete cases, where only some of the answers to the questions that will be asked in a consultation are stored. One way to do this is to say SAVE before the end of the consultation. This will cause only those answers already given to be stored. This can only be done once during a consultation.

Another method is to use the UPDATE option. Since it allows you to delete the value for a parameter, you may remove the answers to questions in the middle of a saved case.

The RENAME Option

The RENAME option can be used for creating new case files that are very similar to stored cases. Enter RENAME and the number of the stored case as special options at the beginning of the consultation. The system summarizes all the information that has been saved on the case file. This proceeds exactly like a consultation run in UPDATE mode except that the system does not stop to ask for updates on parameters that have uncertain values or that have the UPDATE property. After all information has been presented, the user is allowed to change or delete any answers as in UPDATE mode.

8.3. Controlling the Interaction of the Consultant

The TAB Option

The tabular input facility permits entering in "tables" data consisting of several parameters. While this format does not reduce the amount of typing that the user must do to enter the answers, it may speed the interaction by reducing the number of questions, or lumping together several related parameters whose values all come to mind or become known at once.

There are two types of parameter tables: simple tables (members of the auxparmgroup TABLES), and context tables (members of parmgroup CONTEXTTYPES). Simple tables are groups of related parameters from one contexttype. Input to all tables is always on one line and there is only one instance of the table possible per context instance.

The parameters in a context table are the context type's INITIALDATA. Context tables can take input in several lines, one line for each instance of the context type in the consultation. There will, of course, be only one line if the context type has no PROMPT2ND.

Tabular input is selected by including TAB as one of the special options at the beginning of the consultation. Due to the abbreviated nature of tabular input, it is expected that only "experienced" clients will use this mode—little advantage is gained by entering data in a table if you don't understand the table headings (the questions they imply) and the values expected for them.

An example of tabular input from the MYCIN system follows. The user enters one answer per column of the table. An answer may be more than one atom, and may include a certainty factor. Following each answer, the user hits the TAB key (or ↑I) to get to the next column. The user may type * to skip an entry (e.g. it is irrelevant). A vertical bar in the heading indicates a legal termination point. One answer must be entered for each heading up to a vertical bar (or the end of the table). When these bars are used, the heading should explain the circumstances under which remaining parameters may be skipped unless this is obvious. In tables with more than one line, the user may type a quote (") to copy the corresponding answer from the previous line. The user enters DONE to indicate that no more lines of information are needed.

```
--------PATIENT-1--------

Please enter information about the patient.

      Name                          Age        Sex        Race
1) ** PHRED PSMITH                  31         MALE       CAUCASIAN

2) Are there any positive cultures obtained from Phred Psmith?
** Y

        site            date            | method of
                        collected       | collection [urine,sputum only]
  3) ** BLOOD           DEC 8

          identity | stain   morph    grows      tried      | grows
                   |                  aerobic    anaerobic  | anaerobic
                   | [skip this part if identity is known]
    4) ** PSEUDOMONAS-AERUGINOSA
    5) ** U          POS     ROD      Y          N
    6) ** DONE

        site            date collected | coll. method
  7) ** URINE           DEC 10           VOIDED
          identity | stain   morph    grows      tried      | grows
                   |                  aerobic    anaerobic  | anaerobic
                   | [skip this part if identity is known]
    8) ** E.COLI
    9) ** DONE

         site                    date collected | coll. method
  10) ** DONE

11) Are there any pending cultures of Phred Psmith?
** N

. . .

14) Are there any current drugs of Phred Psmith?
** Y

         antimicrobial            route of          date
                                  administration    started
  15) ** PENICILLIN               PO                DEC 9
  16) ** DONE

. . .

Please enter CBC results in the following table

       WBC              Percent          Percent
       in 1000's        PMN              Immature forms
23) ** 15               20               5
```

```
There are several special options available during tabular input:

The following can be used under any heading:

  ?  --  heading will be rephased as a complete question
  *  --  question is not applicable (as opposed to UNKNOWN)
  "  --  copy answer from the line above
  ??, WHY -- same as for normal input

  IMPORTANT:  Use "?" when
        1. you are confused and need to see the full question
        2. you want to see some possible answers


The following should be the first and only entry on the line:

   ASK -- use the standard question asking method for this entire line

   PRINT -- reprint the table thus far as it was interpreted by the
        program [N/A* signifies that the program concluded this answer
        from previously supplied answers, and has ignored your input]

   DONE or NONE -- finished with this section of the table

   HELP -- reviews the above special answers

   CHANGE,STOP,SAVE,COMMENT -- as normally available
```

Figure 8-4: Special Options during TAB

It is possible to nest context tables, as illustrated by the culture and organism tables in the MYCIN interaction above. Thus, for example, when the program tries to trace the ORGANISM context type, it discovers that it must trace the CULTURE context type. At this point a set of tables would be constructed whose layout is roughly:

```
CULTURE-table-1
  ORGANISM-table-1.1
  ORGANISM-table-1.2
  etc.

CULTURE-table-2
  ORGANISM-table-2.1
  ORGANISM-table-2.2
  etc.
```

This behavior would only happen if the system tries to trace a a lower context type. Thus, if the system was trying to trace only the cultures, no nested subtables would appear. The easiest way to force the desired format is to control the setup of contexts. In MYCIN, for example, the bulk of the context tree is acquired at one time during the beginning of the consultation. The goal rule in MYCIN employs a ONCEKNOWN* premise clause to enforce this sequence of context creation.

To define a context table, simply use the Parameter command to edit the definition of the context type, appending the appropriate properties from the list given below. To define a simple table for an

existing context type, again, use the Parameter editor to give the table a name (e.g., BLOODFINDINGS) and make it a member of the auxiliary parameter group TABLES. The following properties will constitute its definition.

TABPARMS a list of the parameters to be acquired in this table. The TABPARMS of a context table must be the INITIALDATA of the context type. The tabular input method does not actually trace a new context's INITIALDATA, so any INITIALDATA parameter not in the table will not be asked until it is subsequently needed in the consult. Each parameter mentioned in the TABPARMS list will automatically have an inverse pointer (BLOCKNAME) added to its definition.

TABHEAD a list of headings, one for each parameter in TABPARMS. Each heading may be a single string, or a list of strings (or NIL); if a list, then the heading is multi-lined, with the elements printed on successive lines.

TABSTOPS a list of tabstops (horizontal character positions), one corresponding to the desired position of each heading, relative to the first heading, which is column 0 (i.e. the first element of TABSTOPS is always zero); plus a final tabstop indicating where the last column of the table ends (position of last heading plus width of last heading/possible answer). The last tabstop thus gives the desired width of the entire table, and is used in positioning the table (if the table is wider than the current indentation permits, it will be shifted left). It is assumed that the tabstops are consistent with TABHEAD, and that in the worst case (i.e. when the table has to be shifted all the way to the left margin) TABHEAD does not overflow a line. Typical line is a maximum of 72-80 columns wide, and the first 6 or so columns of a table are occupied with question numbers and the "**" prompt. These considerations may call for a fair amount of experimentation on the part of the system builder.

The following properties are optional:

TERSEHEAD like TABHEAD, but used when in **terse** mode or refreshing headings (e.g. after an interaction that obscured the table). Ideally should be one line only.

TABPROMPT if present, this is printed as an "introduction" to the table to orient the user, e.g. "Please enter CBC results:". Simple tables generally have TABPROMPTS; context tables generally don't.

LEGALTERM a list of numbers indicating how many answers are legal. Default is (n), where n is the length of TABPARMS, i.e., ordinarily the program expects *all* the columns to be filled in. If there is more than one number in LEGALTERM, however, it is permissible to fill in the table only partially; when the headings are printed, a vertical bar (|) is printed to indicate where it is legal to end the input. Sometimes it happens that the answer to one question implies the answer to another, or implies that the other value is irrelevant. It is this situation which LEGALTERM is intended to handle. E.g., if the user can enter the identity of the organism, then the system can conclude its gramstain, morphology and aerobicity.

TRANS (for TABLES only) translation of the sense of the table; may be used in QA when referring to what the content of a "question" was.

An Example of Tabular Input

The tables shown in the MYCIN interaction above are defined as follows:

```
POSCUL  [CONTEXTTYPES]

  TABPARMS:  (SITE WHENCUL COLLECT)
  TABHEAD:  ("site" ("date" "collected")
                    ("method of" "collection [urine,sputum only]"))
  TABSTOPS:  (0 15 32 62)
  TERSEHEAD:  ("site" "date collected" "coll. method")
  LEGALTERM:  (2 3)

KNOWNORG  [CONTEXTTYPES]

  TABPARMS:  (LABID GRAM MORPH AIRGROW ANTRY ANGROW)
  TABHEAD:  ("identity"
               ("stain" NIL "[skip this part if identity is known]")
               "morph"
               ("grows" "aerobic")
               ("tried" "anaerobic")
               ("grows" "anaerobic"))
  TABSTOPS:  (0 11 19 28 38 50 61)
  TERSEHEAD:  ("identity" "stain" "morph" "aerobic" "tried anaer"
               "anaerobic")
  LEGALTERM:  (1 5 6)

BLOODFINDINGS  [TABLES]

  TABPARMS:  (WBC PMNS BANDS)
  TABHEAD:  (("WBC" "in 1000's")
             ("Percent" "PMN")
             ("Percent" "Immature forms"))
  TABPROMPT:  "Please enter CBC results in the following table"
  TABSTOPS:  (0 15 30 47)
  TERSEHEAD:  ("WBC (1000's)" "% PMN" "% Immature forms")
  TRANS:  (the "results of a recent CBC")
```

The SUMMARY Option

Consultations on stored cases can be run in summary mode by entering SUMMARY and the case number as special options at the beginning of the consultation. As each context is created, the system summarizes the information about that context which the user entered in a previous consultation. Whenever the system would normally ask a question whose answer is stored on the case file, it instead prints out a statement of the value. This provides a much more concise consultation than the standard method of simulating questions and answers when running stored cases.

In order to use the SUMMARY option, each member of CONTEXTTYPES that is to be summarized must be given a CNTXTSUMMARY property. This property tells what parameters should be included in the summaries of contexts of this type. The first element of the CNTXTSUMMARY property is a list of parameters. In order for a context to be summarized when it is created, the case file must include a

stored value for each of these parameters of this context. Inspect the example of the CNTXTSUMMARY property of the POSCUL context type in MYCIN. A summary will be printed for contexts of this type if values are stored for SITE and WHENCUL. If the value of WHENPOS is also available, however, a more informative statement will be printed.

Subsequent elements of CNTXTSUMMARY each represent a group of related parameters to be included in the summary. These elements have one of two forms. First, the element may be a list of pairs representing alternative statements that can be made about the context. The first component of each pair is a list of parameters. If a definite value is stored for each of these parameters of the context, then the second component of the pair will be used as a template for the statement to be printed about the context. The pairs are ordered by decreasing numbers of parameters in the first part; they are examined in order so that the pair which is selected will provide the most information about the context. A statement template consists of text and parameter names; the statement is constructed by replacing each parameter name in the template by its value from the case file. The parameter names must appear in the same order in the list of parameters and in the statement template.

The second element of CNTXTSUMMARY is generally of this form; it represents the identifying statement to be printed when the context is created. [Note that the first element of CNTXTSUMMARY must list all parameters that are named in the last pair of an element of this form.]

```
[(SITE WHENCUL)
 (((SITE WHENCUL WHENPOS)
   (This SITE culture was obtained on WHENCUL , and turned positive
        on WHENPOS .)))
 (((SITE WHENCUL)
 (This positive SITE culture was obtained on WHENCUL .]
```

The second form for an element of the CNTXTSUMMARY property is a header followed by one or more related parameters. The header is a string to be printed describing the group of parameters (or NIL if no heading is desired). For each of these which has a value in the case file, a statement is printed telling its value. Below are two elements from the CNTXTSUMMARY of PATIENT in MYCIN.

```
(NIL MOTHERHOOD PREMATURE JUNKIE CROWD COMPROMISED IMMUNOSUPPRESSED
     ALCOHOLIC DIABETIC DIAGNOSES RENALSUSPECT DIALYSIS)
("Past Medical History:" ALLERS REACTION THER-HIST SURGERY SPLENECTOMY
                         HISTORY NEUROSURG NEUROTIME CNSMALFORM TBRISK
                         ANIMALEXP)
```

The beginning of a MYCIN consultation in summary mode is shown below.

```
13-Nov-78 15:28:35
[consultation of 30-Jan-77 10:15PM]


Pt538 is a 34 years old Caucasian male.
   Pt538 is a compromised host.
   Pt538 is not immunosuppressed.
   Pt538 is not an alcoholic.
   It is suspected that pt538 may have impaired renal function.
```

Recent Medical History:
Pt538 has had recent neurologic signs.
The duration of the neurological signs is .3.
Pt538 has had recent neurologic symptoms.
The duration of the neurological symptoms is 1.

Past Medical History:
Pt538 is not allergic to one or more antibiotics.
Pt538 has not undergone surgery.
Pt538 does not have a tb risk factor.

Physical:
The weight of pt538 is 70.
The height of pt538 is not known.
Pt538 does not have a rash or cutaneous lesions.

CULTURE-1: This pending csf culture was obtained on 30-Jan-77.
A smear of this pending csf culture was examined.

DRUG-1: Cephalothin is currently being administered via iv route.
Therapy began on 23-Jan-77.
It is not known whether the dose of the cephalothin was appropriate.

9. Implementation Details

9.1. Miscellaneous System Variables

The variables below are normally given values automatically in the course of running the context editor. To modify any of these values, use the Variables command.

DOMAIN — a string (enclosed in quotes) describing the domain in which a consultation is to be given, e.g., "infectious disease". Used when telling the user what the consultation system does.

TREEROOT — the root context type, e.g., PERSON.

ROOTID — the value of the PRINTID property of TREEROOT, e.g., PATIENT – .

ROOTPARM — the identifying parameter of the root context, e.g., NAME. This is generally be the first INITIALDATA of the root context type.

ROOTNAMES — a list of ways of talking about the root context during a consultation. It consists of the same word in four formats: uppercase atom, lowercase atom, string with leading and trailing blanks, and short identifier, e.g., (PATIENT patient " patient " Pt). The principal use of the last element is for privacy in storing the case file. If ROOTPARM is NAME, the value of this parameter will be replaced by the short identifier with the case number appended (e.g., patient's name will be stored as Pt135 when the file is saved).

PRINTIDS — A list of all the possible values of PRINTID of a context type, e.g. (PATIENT – ORGANISM – ...). Needed to run a consultation.

TRANSLATOR — is a list of pairs of the form (type (translation)), where type is a parameter group or a context type, and translation is a short description of the type of context. There is one entry for each element of PARMGROUPS except CONTEXTTYPES, e.g. (ORG-PARMS (organism)), and one for each element of CONTEXTTYPES, e.g. (POSCUL (positive culture)). Needed to translate rules.

ALIASES — a list of pairs (contexttype atom) where atom is a short (2 character) atom to use in assigning unique, internal names to contexts of the type contexttype. There is one entry for each element of CONTEXTTYPES, e.g. (POSCUL PC). Used in storing information entered by the user on the case file.

9.2. Pointers to and from Rules

When a rule is defined belonging to a particular rule group, it is added to the list whose Lisp name is that rule group, and the rule's SUBJECT property is set to the name of the group. For example, CULRULES has the value:

```
(RULE292 RULE299 RULE300 RULE570 RULE144 RULE269 RULE234 RULE056 RULE148
RULE579 RULE580 RULE574 RULE560 RULE561 RULE562 RULE564 RULE565 RULE500
RULE501 RULE502 RULE503 RULE504 RULE523 RULE524 RULE525 RULE526 RULE527
RULE160 RULE164 RULE296 RULE297 RULE366 RULE370 RULE396)
```

and the SUBJECT of each of these rules is CULRULES.

The following parameter properties are generated automatically as rules are entered, and are updated when rules are edited or deleted. Each property is a list of rule names. They show which rules use and conclude each parameter. The lists are used in controlling invocation of rules and in question-answering, and are listed here for your information. These lists should *not* be edited manually; this can lead to inconsistency, and any changes you make may be obliterated anyway by the automatic updating mechanism.

(a) these are for consequent rules:

USED-BY — the parameter appears in the PREMISE of these rules; if any of the rules is tried, it may cause this parameter to be traced.

UPDATED-BY — the parameter is updated by the ACTION of these rules; tracing the parameter will cause the rules to be tried.

CONTAINED-IN — the parameter appears in the ACTION, but not because its value is being updated by these rules; tracing the parameter will not cause the rules to be tried.

(b) these are for antecedent rules:

ANTECEDENT-IN — parameter appears in the PREMISE. The rules will be tried as soon as this parameter (and all other parameters in the PREMISE) has been traced. This is analogous to UPDATED-BY, in that this parameter will trigger invocation of these rules.

UPDATED-IN — parameter appears in the ACTION. The rules will conclude about this parameter, but tracing the parameter will not cause the rules to be tried.

(c) these are for mapping rules:

USED-BY-THE-WAY — the parameter appears in the PREMISE of these rules, but it is the parameter of a context different from the one that the rule applies to; e.g. it appears within the predicate of a mapping function and refers to parameter of a context being mapped over.

UPDATED-BY-THE-WAY — the parameter is updated by these rules, but is not the reason for invoking the rule (i.e., tracing this parameter is not what causes the rules to be invoked). Normally not used.

9.3. Internal Data Structures during a Consultation

The variables and dynamic properties listed below may be of some assistance in debugging. What follows assumes that you have some familiarity with Interlisp.

ROOTNODE has as its value the name of the root of the context tree. The name of a context is formed by taking the value of the PRINTID property of its context type and adding a suffix number to indicate the particular occurrence of the context; the name, for instance, of the PERSON context in MYCIN might be PATIENT-1.

CRELST is a list of the names of all the contexts that have been created in the current consultation.

Dynamic Properties of Contexts

The parameters which have a value in a given context are stored as properties of that context; if the patient's name has been asked or otherwise deduced, then NAME would be a property under PATIENT-1. The form of the property, i.e., the result of (GETP '<context> '<parameter>), is the list of values of the parameter in the format (<value> CF) pairs, for example, ((NEUROLEPTIC 800) (TRICYCLIC 600) (MAOINHIBITER 400)). For yes/no parameters, only the value YES is stored, so there is a single pair; a negative CF indicates the value NO.

In addition to parameters, a context has a few other dynamic properties of possible interest. They are:

CRE8 — The type of the context. The type of SYNDROME-1 is SYNDROME.

UPPOINT — The ancestors of the context in the context tree. The UPPOINT of STATE-1 in HEADMED would be (SYNDROME-1 PATIENT-1).

PARMSOF — internal property of no debugging use.

Finally, for each subcontext of the current context, the type of the subcontext is a property; if PATIENT-1 has a subcontext of type, POSCUL, then it also has a property whose name is POSCUL. The value of that property is a list of the form, ((<context-instance-name> 1000) (<context-instance-name> 1000)....), as in ((CULTURE-1 1000) (CULTURE-2 1000)). An entry of this type exists for each and every descendant type below the current context in the tree, regardless of intervening contexts. Example: Suppose that ORGANISM-1 is below CULTURE-1 which is below PATIENT-1. Both PATIENT-1 and CULTURE-1 would have KNOWNORG properties with the value, ((ORGANISM-1 1000)).

Dynamic Properties of Parameters

Parameters have the following dynamic properties:

TRACED — Value is the list of context names in which that parameter has been traced.

ASKED — Value is a list of the contexts in which that parameter has been asked about.

Dynamic Properties of Rules

Rules have the following dynamic properties:

CONCLUSIONS

Value encodes the conclusions that were made using the rule. The CONCLUSIONS property is a list of records, one for each conclusions that the rule made. A record has the following fields:

CVALUE	the value that was concluded
CPARM	the parameter that was concluded
CCNTXT	the context that was concluded
CQNUM	the current question number
CCF	the certainty factor of the conclusion
CSVCNTXT	the context to which the rule was applied (not necessarily the same as

CCNTXT)

CSVTALLY the value of TALLY (set in the PREMISE by $AND)

APPLIEDTO Value tells what contexts a rule was applied to and what the results were. The value is a list of lists. Elements can be of the following format:

(ORGANISM-1)	Rule was successfully applied to ORGANISM-1.
(ORGANISM-1 C1)	Rule was applied to ORGANISM-1 but failed because clause 1 was found to be false.
(ORGANISM-1 C1 T)	Rule was never applied to ORGANISM-1 because clause 1 was known to be false.
(ORGANISM-1 IDENT)	Rule failed due to circular reasoning starting with the parameter IDENT. I.e., some rule caused the system to try to determine the IDENT of ORGANISM-1, and among the rules that were invoked in trying to achieve this goal (or some subgoal) was this rule which used the IDENT of ORGANISM-1. If the rule succeeded, the system would have used the IDENT of ORGANISM-1 in order to conclude (perhaps indirectly) the IDENT of ORGANISM-1.

9.4. Modifying QA for your domain

QUEST is a file containing functions and data structures which can be used to customize EMYCIN's QA module to your consultation system's domain. It contains several variables and functions that you may modify to improve the breadth of the questions asked and coherence of the answers given.

In EMYCIN, all of the QUEST functions return NIL and all the variables are set to NIL. If you wish to take advantage of any of the specializing capabilities, you may redefine the above functions or reset the variables and include the new definitions in your system. Most systems will want to have their own IMPORTANTPARMS, QAUSENAMES, and QAUSELABELS. Only very sophisticated and/or ambitious system builders will want to use any other variables or any of the functions.

QAUSENAMES a list of elements from CONTEXTTYPES that will be used for recognizing references to contexts in questions. If the value of the default parameter for one of these contexts is mentioned (and there is only one context with this value) then we will consider that the value may be used to implicate the context. For example, in "IS THE E.COLI SIGNIFICANT?" "E.COLI" might mean ORGANISM-1.

QAUSELABELS a list of elements; the CAR of an element is a single word describing a group of contexts, and CDR is a list of context types - context types that can be described by that word (e.g. (CULTURE PENDCUL POSCUL)). Note that in most systems there is a single context type in the CDR of an element; the preceding example from MYCIN illustrates the atypical case where there are different types of context that can have the same parameters (viz. pending cultures and positive cultures). For each element, the list of contexts of the indicated types is created. If the list contains a single context, an entry is made into UNIQUELABELS to indicate that when the word is preceded by THE or THIS in a question, it may implicate the context (e.g. "THE CULTURE" may mean CULTURE-1). It is not necessary to add

an entry for your TREEROOT since this is done automatically independent of QAUSELABELS.

IMPORTANTPARMS
used by TEST and EXPLAIN to summarize the results of a consultation. (See section 7.3 for details.)

QAATOMS a list of globals that used elsewhere in the customized QA. Each element consists of a variable that will be used globally, and its initial setting. The CAR of the element is the variable, and the CDR is EVALed to obtain the value that it should be set to. This is done before the user is asked for each question. For example, flags that are set by patterns should be in this list.

QAPATTERNS a list of patterns to augment or override the standard patterns. You should get a copy of the file PATTERNS and copy the format used in that file. PATTERNMATCH will be done first on this set of patterns, then on PATTERNS. Typically these set flags (which are in QAATOMS) to be used to indicate a special type of question (that will be answered by QA.SPECIAL or QA.GENERAL).

QAOPTIONS a list of special options to be recognized before complicated parsing of the questions. If the first word of a question is the CAR of some element in the list, then the CDR of that element is EVALed to generate the answer to the question. This assumes that you are writing your own functions to take care of special options.

QA.PATTERNFIX called after both sets of patterns have been applied to do any necessary fixing up. Typically this sets defaults for flags that weren't set by patterns.

QA.SPECIALINITFN
Function that is called immediately upon entry into QA to do initialization that is necessary for any specialized functions that you may have written.

QA.GENERAL Function that is called before standard question-answering routines to answer a general question (i.e. not specific to a consultation). Checks to see if this is really a domain-dependent question. Returns T if it answers the question.

QA.SPECIFIC Function that is called before standard question-answering routines to answer a question that is specific to a consultation. Checks to see if this is really a domain-dependent question. Returns T if it answers the question.

9.5. The Dictionary

For those who wish to make maximal use of the dictionary, the format of the dictionary and maintenance routines are described below. A dictionary can be created using the Words command. You can get a copy of the dictionary by using the dictionary option of the Knowledge-base-lister command.

The default dictionary does not know about synonyms, and the pointers that associate English words with parameters are fairly crude, based simply on the words in each parameter's TRANS property initially.

You can edit the dictionary by using the Edit-word option of Words command. It prompts for words and tells you what it knows about them. For any new word, you can make it a synonym of an existing

word; or make it its own synonym by simply typing <escape> (root words are their own synonyms. The system will complain about root words with no other props, but you can just disconfirm any attempt to delete such words). You can also change the synonyms of existing words.

The Concept-editor option of the Words command is a routine specifically designed to edit the CONCEPT properties of each parameter. The CONCEPT prop tells which English words strongly or weakly implicate it (see below), and is very significant to the QA module's parsing routines.

Dictionary Word Descriptors

The following properties of words reside in the dictionary. Most are used in understanding English text (primarily in QA).

NOCONTENT — the word is content-free for QA's purposes (has none of the above properties; useful only in pattern matching)

TXTSYN — synonym when the word appears in free text, as opposed to in response to a question. TXTSYN's are restricted to 4 classes of equivalence:

1. definitional
   TXTSYN of HOSPITAL-ACQUIRED is NOSOCOMIAL
2. abbreviational
   TXTSYN of MORPH is MORPHOLOGY
3. same root (but irregular, so the English root word extractor fails)
   TXTSYN of BEGUN is BEGIN
4. pattern matching textual synonyms
   TXTSYN of DISCUSS is USE
   (this class reduces the list of patterns)

VALUESYN — synonym of the word when used as an answer to a question during the consultation. This property is explicitly specified only when it differs from the word's TXTSYN. The two may differ if the special circumstances of a question during a consultation gives a different meaning to the word.

Canonical example from MYCIN:

NEGATIVE has VALUESYN GRAMNEG, but
NEGATIVE has TXTSYN of NEGATIVE

COMPOUND — Marks words which might be combined with the words following in input stream. There are 3 possible forms, as follows (the first two types are generated automatically):

1. word is the first part of a hyphenated word; COMPOUND property specifies maximum number of pieces that may follow, plus the words which may result;
   e.g. PROTEUS: (2 PROTEUS-MIRABILIS PROTEUS-NON-MIRABILIS).
2. word is used to denote context names, and may be joined with following number. COMPOUND property is the atom CONTEXT.
   e.g. INFECTION: CONTEXT.

3. word is part of a 2-word phrase that can be replaced by a single word; COMPOUND property is a list of pairs (2nd-word replacement).
e.g. HOW: ((OLD AGE)) ["HOW OLD" is replaced by "AGE"]

The following properties are generated automatically from other information.

EXPECTED list of parameters for which the word is a legal value

INFUNCS list of predicates; the word appears in TRANS of these functions

INCONCEPT list of parameters; the word appears in the CONCEPT property of these parameters

TXTSYNOF list of words whose TXTSYN is this word example:

BEGIN has TXTSYNOF: (BEGUN)

VALUESYNOF analogously for VALUESYNs.

SPEC list of parameters; word is an ambiguous value of these parameters and in the parameters' SPECIAL property.

HYPART the word is part of a hyphenated word, but not the first part; value of the HYPART prop is a list of words of which it is a part. Its TXTSYN must be itself. e.g. the HYPART of TISSUE is (LUNG-TISSUE RENAL-TISSUE).

PATWORD True if word appears in a pattern; such words must always be their own TXTSYNs. This property is there only to make sure nobody accidentally changes the TXTSYN.

Dictionary Properties of Parameters

CONCEPT A list ((strong-words) weak-words...) indicating which words in the dictionary strongly or weakly implicate it. The name of the parameter itself (usually) is a strong word. When the dictionary is first built, a CONCEPT property is automatically generated from the parameter's TRANS, but you are free to refine it if you wish. Use the concept-editor from the Words command.

MUSTMATCH Indicates words or combination of words which *must* appear in a question in order to implicate this parameter. The value of the property is a list of lists, each sublist interpreted as a simple pattern, one of which must be matched by the input for this parameter to be implicated (i.e. each word in the pattern must be present in the sentence). e.g.: CSFPOLY: ((CSF PMN)) The ordering of words in the pattern is unimportant.

RULE.OUT Indicates parameters ruled out by this parameter. That is, if this a parameter is implicated, then the parameters in its RULE.OUT property are no longer considered. e.g. ENTEROBACTERIACEAE: (ORGANISMS)

INTERNAL if T, no question will ever be interpreted to be about this parameter.

Enter MUSTMATCH, RULE.OUT and INTERNAL props via the parameter editor.

9.6. Defining New Predicate and Action Functions

The predicates and actions supplied with EMYCIN cover most simple relationships on associative triples, but there are times you may need additional functions for your domain. The following example should be followed when defining new functions which you want EMYCIN to know about. This will

enable EMYCIN to check the type and consistency of the arguments for a specific function call, produce appropriate translations, update the necessary rule pointers, etc. In this section we, of course, assume you are able to program in Lisp.

In the following example, taken from the SACON system, the knowledge engineer defines UNIQUEP, a predicate that tests whether the value of a multi-valued parameter is known uniquely, i.e., that there is only a single value known in the current context.

```
Rules, Parms, Go, etc.? Parameters

Parameter name: UNIQUEP
Group for new parm UNIQUEP : FUNCTIONS
UNIQUEP is not yet defined; do you want to define it now? Yes
**(DEFINEQ (UNIQUEP (CNTXT PARM)
**              (PROG (VALUES)
**              (* Determine the values of parm; system will infer if needed)
**              (* VAL returns a list of (value CF) pairs)
**              (SETQ VALUES (VAL CNTXT PARM))
**              (RETURN (AND VALUES (NOT (CDR VALUES)]

adding to CHANGES
(UNIQUEP)

Is UNIQUEP a premise function? Yes

TRANS: USE KNOWN
Copying TRANS of KNOWN:
edit
[(COND [(YNPARM (# 2))
        '(it is known whether ((2 1]      See below for an explanation
       (T '(((2 1))                       of these translation expressions
            is known]
*(1 ((2 1)) "is known uniquely")          The user looked at the TRANS for
                                          KNOWN to refresh his memory.
*OK
UNIQUEP
TEMPLATE: (CNTXT PARM)
Enter any more props now for UNIQUEP ? Yes

Property: TRSTRANS
UNIQUEP
TRSTRANS: USE KNOWN
Copying TRSTRANS of KNOWN:                TRSTRANS is used for terse translations
edit                    (TRSTRANS)
((2) is known)
*(N uniquely) P
((2) is known uniquely)
*OK
Property:<cr>

Parameter name:<cr>

Rules, Parms, Go, etc.? Save
```

You should give your function a definition by dropping into Lisp, using DEFINEQ as above, and then return to the EMYCIN executive. This definition should use the primitive functions described in appendix I, which deals with EMYCIN rule functions. The functions VAL, VAL1, SAME, and CONCLUDE are commonly used. When you define the new function the system will record it as intended for the file CHANGES.

The regular parameter editor is used to inform the system of the function name. User-defined functions are members of the auxiliary parameter group FUNCTIONS. The system then prompts you to enter the properties TEMPLATE and TRANS. TEMPLATE looks like the argument list of the function. It is used primarily to perform type checking of the functions arguments when used in rules. Each entry in the TEMPLATE list tells EMYCIN the expected type of object for that argument in the function call. There are numerous tokens that are used to identify the types EMYCIN expects; these are listed below.

CNTXT	a context variable, normally CNTXT itself.
PARM	a parameter of some context type.
PARMS	a list of parameters (need not be QUOTED).
TABLE	a table of parameters, i.e. a member of TABLES.
VALU	a value of the parameter in this clause.
VALUS	a QUOTEd list of (value CF) pairs for this parameter.
CF	an integer in the range (-1000, 1000) or a numeric expression.
TALLY	a tally variable, typically TALLY. Can also be a CF.
TEXT	a string or a TEXTAG call. (see appendix section I.5) about text functions.
IGNORE	ignore the contents of this argument

TRANS is a list used to translate a call to the function, and has expressions that are significantly different from the TRANS properties of normal parameters. In particular, they contain expressions for selecting various arguments that are to be translated. For example, a TRANS for the function GREATERP* might be "((1) is greater than (2))". Below is listed the complete documentation for function TRANS properties. You could look at the TRANS of various system predicates (on the list SYSFUNCTIONS) for examples.

Pattern elements in the TRANS of a function are interpreted as follows:

{non-list} translates as self.

(n1 n2 ...) list of numbers, interpreted as edit commands to descend into the arg list; translates the designated element, e.g. most common case: (n) translates the n'th argument.

(# n1 n2 ...)
similar, but simply selects the designated arg without translating it.

((n m) . whetherflg)
is used to translate a phrase "the {nth arg} of {mth arg}", i.e. "parameter of context." WHETHERFLG means to prefix the word "whether" if parameter is yes/no, to make a subordinate clause out of it. Standard case is just ((2 1)) used to translate VAL and VAL1.

(FUNCTION func)
: func is applied to ARGS to produce a translation.

(NLAM --), (SAMETRANS --), (SAME --)
: internal calls to special translating routines (for NLAMBDA nospreads (e.g. $AND), SAME-type predicates, and terse translation of SAME functions, respectively).

(INFIX trsop verboseop)
: translates an arithmetic function using infix operator. Also parenthesizes translation if the PRECEDENCE property of the current function is lower than that of the parent form.

{other list}
: is EVALed, and the result is treated as a new PATTERN to apply. If the result is a non-list, or a quoted list, however, it is taken as a translation, rather than a pattern. An example of this can be seen in the TRANS for KNOWN above.

Appendix I
Functions

This appendix describes in detail the different types of functions that can appear in rules. Predicate functions make up clauses in a rule premise. Conclusion functions appear in the action. Auxiliary functions are miscellaneous functions that are not predicates, but may be used for computing values in a premise or action. Text functions provide extra support for TEXT – valued parameters. Mapping functions allow you to perform some operation on a whole set of contexts or values.

I.1. Non-Numeric Predicate Functions

When any of these predicates is used in the PREMISE of a rule, the argument CNTXT should be filled in with the atom CNTXT, not the name of any context or context type. The exception to this occurs within the predicate of a mapping function as discussed later. The argument PARM is filled in with a parameter name, PARMS with a quoted list of parameter names. VALU is usually filled in with a legal value of the PARM:

(SAME CNTXT SITE BLOOD) – The site of the culture is blood.

It is also possible to fill in VALU with a Lisp form that when evaluated, returns the name of a value (e.g., a call to VAL1), a list of values (e.g., a call to ONEOF or LISTOF), or a list of (value CF) pairs (e.g., a call to VAL or VALYEW). See the Auxiliary functions section (I.4) for more details.

If the PARM is a yes/no parameter, you should omit VALU altogether, for example:

(SAME CNTXT COMPROMISED) - The patient is a compromised host,
(NOTSAME CNTXT BURNED) - The patient has not been seriously burned.

The predicates SAME and THOUGHTNOT return CF's (internally represented as integers from – 1000 to 1000) to be minimized by $AND. This determines the setting of TALLY (that is used in the rule's ACTION), see section 4.2 for details. The other predicates return true (T or 1000) or false (NIL). It is likely that your rules will only need a few of the functions described below, probably just SAME, NOTSAME, KNOWN, and NOTKNOWN. These functions are most conveniently entered in ARL.

The following 9 functions test whether the value of PARM is or is not VALU, with varying shades of certainty. SAME and NOTSAME are the most commonly used (they are logical negations of each other), but a whole spectrum is available, best visualized as follows as a function of the certainty that PARM of CNTXT is VALU:

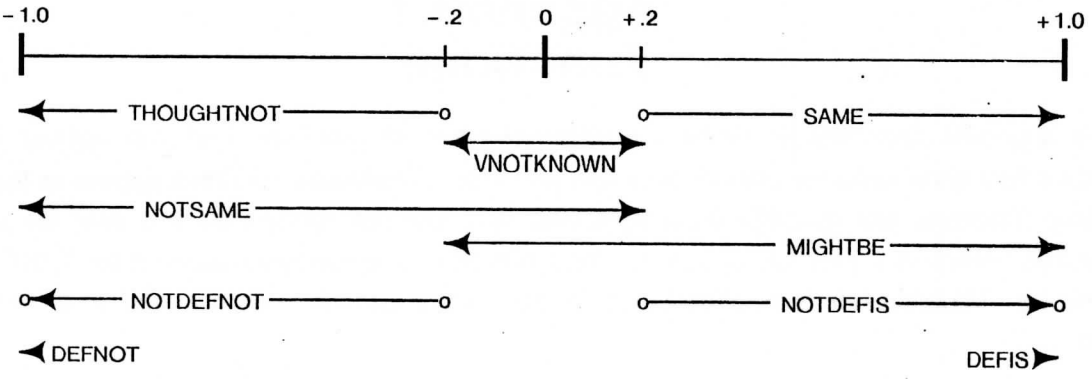

Figure 9-1: Non-Numeric Predicate Functions

(SAME CNTXT PARM VALU)

Checks to see if PARM of CNTXT is VALU, returning the associated CF. True if the CF is greater than .2.

```
Ex: (SAME CNTXT SITE BLOOD)
    -> The site of the culture is blood.
```

ARL form: (SITE = BLOOD)

(NOTSAME CNTXT PARM VALU)

The logical compliment of SAME, returns true if PARM of CNTXT is not thought to be VALU (CF $\leq$ .2).

```
Ex: (NOTSAME CNTXT SPECSTAIN)
    -> Organisms were not seen on the stain of the culture.
```

(MIGHTBE CNTXT PARM VALU)

True if PARM of CNTXT might be VALU, i.e. there is no evidence against it (CF > - .2).

```
Ex: (MIGHTBE CNTXT ADEQUATE)
    -> There is no evidence that the dose of the drug was not
       appropriate.
```

(THOUGHTNOT CNTXT PARM VALU)

Checks to see if PARM of CNTXT is not VALU, i.e., there is evidence against it, returning true if the associated CF is less than - .2. The value returned is the negative of that CF. This is the algebraic negation of SAME, whereas NOTSAME is the logical negation.

```
Ex: (THOUGHTNOT CNTXT IDENT E.COLI)
    -> There is evidence that the identity of the organism is not
       E.coli.
```

(VNOTKNOWN CNTXT PARM VALU)

Returns true if it is not known whether the value of PARM of CNTXT is (or is not) VALU (- .2 $\leq$ CF $\leq$ .2).

```
Ex: (VNOTKNOWN CNTXT IDENT E.COLI)
    -> It is not known whether the identity of the organism is E.coli.
```

(DEFIS CNTXT PARM VALU)

Returns true if PARM of CNTXT is known with certainty to be VALU (CF = 1.0).

```
Ex: (DEFIS CNTXT IDENT MYCOBACTERIUM-TB)
   -> It is definite that the identity of the organism is
      Mycobacterium-tb.
```

(DEFNOT CNTXT PARM VALU)

Returns true if PARM of CNTXT is definitely not VALU (CF = −1.0).

```
Ex: (DEFNOT CNTXT IDENT VIRUS)
   -> It is definite that the identity of the organism is not Virus.
```

(NOTDEFIS CNTXT PARM VALU)

Returns true if PARM of CNTXT is thought to be VALU, but not with certainty (.2 < CF < 1.0).

```
Ex: (NOTDEFIS CNTXT IDENT CRYPTOCOCCUS)
   -> It is suspected that the identity of the organism is
      cryptococcus.
```

(NOTDEFNOT CNTXT PARM VALU)

Returns true if PARM of CNTXT is thought not to be VALU, but not with certainty (−1.0 < CF < −.2).

```
Ex: (NOTDEFNOT CNTXT IDENT E.COLI)
   -> It is suspected that the identity of the organism is not E.coli.
```

The following 4 predicates test how certain the system is of the value of PARM, independent of what the value actually is.

(DEFINITE CNTXT PARM)

Returns true if PARM of CNTXT is known with certainty (CF = 1.0; for yes/no parameters, also CF = −1.0).

```
Ex: (DEFINITE CNTXT IDENT)
   -> The identity of the organism is known with certainty.
```

(KNOWN CNTXT PARM)

Returns true if the value of PARM of CNTXT is known (CF > .2; for yes/no parameters, also CF < −.2).

```
Ex: (KNOWN CNTXT IDENT)
   -> The identity of the organism is known.
```

(NOTDEFINITE CNTXT PARM)

Returns true if PARM of CNTXT is not known with certainty (CF < 1.0; for yes/no parameters, −1.0 < CF < 1.0).

```
Ex: (NOTDEFINITE CNTXT GENUS)
   -> The genus of the organism is not known with certainty.
```

(NOTKNOWN CNTXT PARM)

Returns true if PARM of CNTXT is not known (CF ≤ .2; for yes/no parameters, −.2 ≤ CF ≤ .2).

```
Ex: (NOTKNOWN CNTXT IDENT)
   -> The identity of the organism is not known.
```

The following two predicates behave somewhat like KNOWN, but are typically used for explicit control of parameter tracing or context setup.

(ONCEKNOWN CNTXT PARM RETFLG)

Finds the value of PARM of CNTXT. If RETFLG is NIL, it means "you have found a value for PARM of CNTXT"; returns the same as KNOWN would. If RETFLG is T, it means "find out all you can about PARM of CNTXT"; this causes tracing, but ONCEKNOWN will return true even if nothing was found out.

```
Ex: (ONCEKNOWN CNTXT SAMEBUG)
   -> There is an organisms with possibly the same identity as this
      organism.

    (ONCEKNOWN CNTXT SAMEBUG T)
   -> An attempt has been made to deduce the organisms with possibly
      the same identity as the organism.
```

(ONCEKNOWN* CNTXT PARMS)

Traces each of PARMS of CNTXT and returns true regardless of the result of this tracing. This is like calling ONCEKNOWN repeatedly, once for each parameter in PARMS, with RETFLG set to T in the calls.

```
Ex: (ONCEKNOWN* CNTXT '(CURTHER PRIORTHER))
   -> Information has been gathered about current drugs of the patient
      and prior drugs of the patient.
```

Note: As shown in the example, PARMS can refer to context types lower in the context tree. These context types (and any intermediate context types) will then be set up by ONCEKNOWN*.

I.2. Numeric Predicate Functions

There are five numeric predicate functions to be used with parameters that take on numeric values (a parameter with DATE for its EXPECT property accepts a date as input, but internally stores the answer as the number of days ago (relative to the date of the consultation), so also uses these functions). The Lisp functions PLUS, DIFFERENCE, MINUS, TIMES, FQUOTIENT, FIX and EXPT may be used to form numeric expressions to be used in these predicates, or as computed values in conclusions. They are defined in EMYCIN rules to return NIL if any of their arguments are NIL; hence you can use VAL1 on possibly unknown parameters without error.

The translations of numeric expressions can be very wordy. To have an expression translated tersely (using arithmetic operator symbols instead of text), enclose the expression in a call to the function TRSEXP . For example, the following expression is found in a SACON rule:

```
      (FQUOTIENT (VAL1 CNTXT WEIGHT)
                 (TIMES 2 (VAL1 CNTXT LENGTH)
                          (VAL1 CNTXT THICKNESS)
                          (VAL1 CNTXT DENSITY)))
   -> the weight of the sub-structure divided by [2 times the length
      of the sub-structure times the wall thickness of the
      sub-structure times the density of the material].
```

TRSEXP greatly improves readability:

```
      (TRSEXP (FQUOTIENT (VAL1 CNTXT WEIGHT)
                         (TIMES 2 (VAL1 CNTXT LENGTH)
                                  (VAL1 CNTXT THICKNESS)
                                  (VAL1 CNTXT DENSITY))))
```

```
-> Weight / (2 * length * thickness * density).
```

The arguments to the five predicates listed here must be numbers, or numeric expressions formed by the Lisp arithmetic operators with the values of numeric parameters, standardly accessed by the function VAL1. Since VAL1 returns NIL if its parameter is unknown, the convention of the EMYCIN rule evaluator is that the 6 Lisp functions mentioned above, plus the 5 predicates listed below all return NIL if any of their arguments are NIL. Thus any of these predicates will simply fail if one of the VAL1's embedded anywhere inside fail.

Note that ARL provides a much more convenient means of entering most of these functions; e.g. the TRSEXP output above could have been used for input.

(BETWEEN* VALU LLIM ULIM)

True if LLIM $\leq$ VALU $<$ ULIM

```
Ex: (BETWEEN* (VAL1 CNTXT AGE) .5 10)
   -> The age of the patient is between 6 months and 10 years.
```

(GREATEQ* X Y)

True if X and Y are numbers and X $\geq$ Y.

```
Ex: (GREATEQ* (VAL1 CNTXT NUMPOS) 2)
   -> The number of cultures from this site which were positive for
      this organism is greater than or equal to 2.
```

(GREATERP* X Y)

True if X and Y are numbers and X $>$ Y.

```
Ex: (GREATERP* (VAL1 CNTXT CSFGLUC) 80)
   -> The csf glucose value is greater than 80.
```

(LESSEQ* X Y)

True if X and Y are numbers and X $\leq$ Y.

```
Ex: (LESSEQ* (VAL1 CNTXT WEIGHT) 70)
   -> The weight of the patient is less than or equal to 70 kgms.
```

(LESSP* X Y)

True if X and Y are numbers and X $<$ Y.

```
Ex: (LESSP* (VAL1 CNTXT CSFGLUC) 80)
   -> The csf glucose value is less than 80.
```

(UNITS NUM UNIT)

Returns NUM, but when translated includes the indicated units. This is needed *only* for those occasions when the correct units are not obvious; "obvious" is defined as being associated with a single parameter that has a UNITS property and appearing in one of the 5 predicates above, or in a CONCLUDE. All the examples shown above are "obvious"; note that those parameters having units were translated correctly without the need for UNITS.

```
Ex: (LESSP* (DIFFERENCE (VAL1 CNTXT WHENSTART)
                        (VAL1 CNTXT WHENSTOP))
            (UNITS 2 DAYS))
   -> The time since the therapy with the drug was started minus the
      time since therapy with the drug was discontinued is less than
      2 days.
```

I.3. Conclusion Functions

Each function in a rule's ACTION concludes about one or more context-parameter-value triples. The strength of each conclusion is indicated by a CF for each triple in the conclusion clause. The CF's will be computed using the certainty of the rule's PREMISE, the CF for the rule itself, and the results of previous rule conclusions of the same triple. See section 4.2 for the details of this computation.

For the conclusion functions described below, the argument CNTXT should be filled in with the atom CNTXT (except in mapping rules). The argument TALLY is always filled in with the atom TALLY. PARM is filled in with a parameter name, and VALU with a legal value for the parameter, or a form that evaluates to the desired value (*not* a list of values; use CONCLUDE* for that). The argument CF should be filled in with the desired certainty factor (a number, though it is also possible to have a form that evaluates to a CF), with a negative CF to indicate a negative conclusion. While CF's are logically treated as numbers in the range [-1,1], a CF is represented internally as an integer from -1000 to 1000, and it is this form that you must use (however, if you use ARL, you can type a "natural" CF and it will be appropriately converted).

Note that, unlike with SAME and related predicates, if the parameter in the conclusion function is a yes/no parameter, the VALU must be filled in. For a positive conclusion the value is YES, for a negative it is NO (which is the same as YES with a negative CF). E.g.

```
        (CONCLUDE CNTXT CONTAMINANT NO TALLY 800)   or
        (CONCLUDE CNTXT CONTAMINANT YES TALLY -800)
    ->  There is strongly suggestive evidence (.8) that the organism is
        not a contaminant.
```

(CONCLUDE CNTXT PARM VALU TALLY CF)

Concludes that PARM of CNTXT is VALU. The conclusion made by this call will have a CF that is TALLY "times" CF.

```
Ex: (CONCLUDE CNTXT IDENT E.COLI TALLY 400)
    -> There is weakly suggestive evidence (.4) that the identity of
       the organism is E.Coli.
```

(CONCLUDE* CNTXT PARM TALLY VALUS)

Performs multiple CONCLUDE's for a single CNTXT and PARM. VALUS is a quoted list of pairs (value CF); each value is concluded with the corresponding CF.

```
Ex: (CONCLUDE* CNTXT IDENT TALLY '((E.COLI 400)
                                   (KLEBSIELLA-PNEUMONIAE 300)
                                   (PROTEUS-MIRABILIS 300)))
    -> There is evidence that the identity of the organism is e.coli
       (.4) klebsiella-pneumoniae (.3) proteus-mirabilis (.3).
```

Note that there exist convenient ARL forms for CONCLUDE, CONCLUDE*, and CONCLUDET, see figure 4-5.

(CONCLUDET CNTXT SWITCH CASES TALLY PARM VALUS)

"Tabular" rule concluding function. Concludes that PARM of CNTXT is one or more of the values in VALUS, according to the value of SWITCH.

SWITCH is a form to EVAL which must return a value, typically a number. It is generally a call to VAL1 for some parameter.

VALUS is a quoted list (VAL1 VAL2 ... VALn) of values for PARM.

CASES is a quoted list of cases that test SWITCH and supply CF's for the values in VALUS. Only one case should be true for any given SWITCH; for the case that is true, the values in VALUS are concluded with the corresponding CF's. Possible cases are:

(EQ valu CF1 CF2 ... CFn)	equal, i.e. succeeds when SWITCH = VALU
(LT num CF1 CF2 ... CFn)	less than, i.e. succeeds when SWITCH < NUM.
(BT num1 num2 CF1 CF2 ... CFn)	between: succeeds when NUM1 ≤ SWITCH < NUM2.
(GE num CF1 CF2 ... CFn)	greater or equal: succeeds when NUM ≤ SWITCH.
(U CF1 CF2 ... CFn)	unknown: succeeds when SWITCH = NIL.

Each CFi must be included in each case; if a particular value doesn't apply, the corresponding CF is 0. In addition, if no "U" case is supplied, the premise of the rule should contain a clause or clauses that ensure that the SWITCHUM expression is known, so that the rule doesn't "succeed" when there is actually nothing to conclude. In the case of SWITCHNUM = (VAL1 CNTXT parm), the desired premise clause would be (KNOWN CNTXT parm).

```
Ex: (CONCLUDET CNTXT (VAL1 CNTXT LENSIGN)
               '((LT 13 -400 -500)
                 (BT 13 20 -500 -400)
                 (GE 20 600 300))
               TALLY TYPE '(BACTERIAL VIRAL))
 -> The type of the infection is as follows:
    If the duration of the neurological signs is:
      a) less than 13 days then: not bacterial (.4), not viral (.5);
      b) between 13 days and 20 days then: not bacterial (.5), not
         viral (.4);
      c) greater or equal to 20 days then: bacterial (.6),
         viral (.3).
```

(DO-ALL CLAUSE1 CLAUSE2 ...)

For multiple conclusions – evaluates each of its arguments (rule conclusions).

(DONTASK CNTXT PARM)

Conclusion function which says that PARM of CNTXT should not be asked (although it may be traced, if needed).

```
Ex: (DONTASK CNTXT CONFORM)
 -> Don't ask about the growth conformation of the organism.
```

(NOTRELEVANT CNTXT PARMS CF)

A conclusion function that indicates that the value of each of the parameters in PARMs is not relevant for CNTXT: we shouldn't ever ask or try rules to deduce the value.

```
Ex: (NOTRELEVANT CNTXT '(SECONDARY) 1000)
 -> It is definite (1.0) that the following is irrelevant: the
    infection to which the bacteremia is secondary.
```

Of course, any rules that make use of DONTASK or NOTRELEVANT must be invoked *before*

any attempt is made to trace the parameters mentioned.

(PRINTCONCLUSIONS CNTXT PARM HEADER EVENIFNIL)

Displays nicely the value of PARM of CNTXT; if no conclusions were made, prints a message to that effect only if EVENIFNIL was set. Intended as a simple goal rule ACTION. If HEADER is T, prefixes values with a simple header announcing what these are the values of; if other non-NIL value, HEADER is printed; otherwise no header at all is printed. Since PRINTCONCLUSIONS does not make any conclusions, rules with this function in their action will *not* be tried in an attempt to trace the parameter. In order to get the rule invoked at all, then, it must be an ANTECEDENT rule which will be tried when the parameters in its action are all traced.

```
Ex: (PRINTCONCLUSIONS CNTXT REGIMEN T)
   -> Display the therapeutic regimen of the patient.
```

I.4. Auxiliary functions

A number of auxiliary functions that are not themselves predicates are used within the PREMISE and ACTION of rules.

($AND CLAUSE1 CLAUSE2 ...)

True if *all* of the clauses are true. The CF returned is the minimum CF returned by any clause. Evaluates the clauses in order until one fails. All rule PREMISEs and the predicates of all mapping functions must be calls to $AND, even if there is only a single predicate clause inside the $AND.

($OR CLAUSE1 CLAUSE2 ...)

True if *any* of the clauses are true. The CF returned is the maximum CF returned by any clause. Evaluates the clauses in order; stops if one of the clauses evaluates with certainty (since the maximum 1.0 is the result).

```
Ex:  ($OR (GREATERP* (VAL1 CNTXT WBC) 12.5)
          (GREATERP* (VAL1 CNTXT PMNS) 80)
          (GREATERP* (VAL1 CNTXT BANDS) 10))
   -> 1) The white count from the patient's peripheral CBC (in
                 thousands) is greater than 12.5, or
      2) The percent of PMN's in the CBC is greater than 80, or
      3) The percent of peripheral WBC's which are immature in the CBC
         is greater than 10.
```

(LISTOF LISTNAME)

Simply evaluates its argument; translates as "one of: <list>". Usually its argument is the name of a list (which without this function call would have been left unevaluated), but it can also be some expression which will return a list.

```
Ex: (SAME CNTXT SITE (LISTOF STERILESITES))
   -> The site of the culture is one of: those sites that are normally
      sterile.
```

Note: these lists should be defined and given TRANS's etc. via the "Variables" command.

(ONEOF X1 X2 ...)

A "no-spread quote": returns its argument list. Used as argument to a basic predicate when a choice of values is indicated.

```
Ex: (SAME CNTXT SITE (ONEOF URINE SPUTUM))
    -> The site of the culture is one of: urine sputum.
```

(QUOTE X)
Lisp function that returns its argument unevaluated. This function is necessary in rules only to quote lists (which would otherwise be interpreted as Lisp forms to be evaluated), e.g. a list of parameters, values, etc. You can type this as 'X, rather than (QUOTE X).

(VAL CNTXT PARM)
Returns PARM of CNTXT as a list of (value CF) pairs, sorted by decreasing CF. VAL traces the parameter if it has not been traced yet.

(VAL1 CNTXT PARM)
Returns the most highly confirmed value of PARM of CNTXT (i.e. the value with the highest CF; the value is returned alone, without the CF). VAL1 traces the parameter if it has not been traced yet. Only makes sense for single-valued parameters, and is generally only used to obtain the value of a parameter whose value is always known with certainty.

(MEASURE1 CNTXT PARM)
Like VAL1, but returns the highest CF, rather than the value: "the measure of certainty associated with <parm> of <cntxt>". Provided for those rare cases where you want to directly manipulate a CF, e.g. to provide a computed CF in a conclusion, or test one explicitly in a premise.

I.5. Functions for TEXT-valued Parameters

The conclusion function CONCLUDETEXT is used for TEXT-valued parameters (viz., those whose LEGALVALS is TEXT). This function is the same as CONCLUDE, but its translation is more appropriate for TEXT-valued parameters. Such parameters are typically goal parameters, used to conclude the "results" or "recommendation" of the consultation.

TEXT-valued parameters take arbitrary pieces of text as their values. A text string may be used explicitly as the VALU argument of CONCLUDETEXT. It is often the case, however, that the same text string is concluded in a number of rules. When this is so, it is convenient to define a TEXTAGS whose TRANS is that text string. This atom can then be used with the function TEXT in place of the actual string to fill in the VALU argument to CONCLUDETEXT in the rules. For example,

```
$OVERINFL [TEXTAGS]
    TRANS:  ("Elevated lung volumes indicate overinflation.")

      (CONCLUDETEXT CNTXT FINDINGS.OAD (TEXT $OVERINFL) TALLY 1000)
   ->  It is definite (1.0) that the following is one of the findings
       about the diagnosis of obstructive airways disease:
       Elevated lung volumes indicate overinflation.
```

In some cases, part but not all of a text string is common to the conclusions of many rules. When this is so, it is possible to define a TEXTAGS with the common sub-string as its TRANS. The rules can construct the entire string using the functions TEXT and TEXTAG. For example,

```
$GETRESULTS [TEXTAGS]
   TRANS:  ("It is imperative to obtain the result of")
```

```
(CONCLUDETEXT CNTXT RECOMMENDATIONS (TEXT 1 (TEXTAG $GETRESULTS)
                                            "an EMI or brain scan.")
              TALLY 1000)
-> It is definite (1.0) that the following is one of the
   recommendations for the patient:  It is imperative to
   obtain the result of an EMI or brain scan.
```

The function PRINTCONCLUSIONS has a special facility for use with TEXT-valued parameters: if the parameter has a property LABEL.ORDER, and values for the parameter were concluded using the function TEXT, then values of equal CF will be sorted by their labels. For LABEL.ORDER = T, the labels are integers (e.g., RECOMMENDATIONS in the last example would have its LABEL.ORDER property equal to T) and the values will be sorted in ascending order of label; otherwise, the LABEL.ORDER property is a list of labels (atoms), and the values are sorted according to the order of the labels in this list. For example, consider the parameter FINDINGS.OAD (from PUFF) that was used in the first example above:

```
FINDINGS.OAD  [PT-PARMS]

   TRANS:  ("the findings about the diagnosis of obstructive
            airways disease")
   MULTIVALUED: T
   LEGALVALS:  TEXT
   LABEL.ORDER: ($OVERINFL $FEV1/FVC4 $FEV1/FVC3 $FEV1/FVC1
                 $FEV1/FVC2 $MMF/FVC1 $MMF/FVC2 $MMF/FVC3 $MMF $SLOPE
                 $DILATION1 $REV $REVMMF $REV1 $REV2 $REV3 $REV4)
```

The atoms in the LABEL.ORDER list are labels and "text tags" (members of TEXTAGS). The earlier example showed a conclusion with a text tag ($OVERINFL); the example below uses a label instead.

```
(CONCLUDETEXT CNTXT FINDINGS.OAD
              (TEXT $FEV1/FVC2
                  "Forced Vital Capacity and the FEV1/FVC ratio
                   are reduced, suggesting airway obstruction of a"
                   (VAL1 CNTXT DEG_FEV1) "degree.")
               TALLY 1000)

-> It is definite (1.0) that the following is one of the findings
   about the diagnosis of obstructive airways disease: Forced
   Vital Capacity and the FEV1/FVC ratio are reduced,
   suggesting airway obstruction of a <deg_fev1> degree.
```

Notice that the label ($FEV1/FVC2) has no translation. It is not a text tag; its only function in the conclusion is to control the order in which many conclusions are printed by PRINTCONCLUSIONS.[15]

It is the convention (though not strictly necessary) that labels and text tags begin with the character "$". This is to avoid the inadvertent use for some other purpose of an atom which is already a text tag or label. Text tags are members of TEXTAGS and have TRANSes; labels do not have TRANSes, and are not members of TEXTAGS.

---

[15] Warning: One should avoid using the text conclusions of any rule in the premise of other rules. The function SAME will not find that (SAME CNTXT FINDINGS.OAD $FEV1/FVC2) is true.

The functions CONCLUDETEXT, TEXTAG, and TEXT are described below.

(CONCLUDETEXT CNTXT PARM VALU TALLY CF)

This function calls CONCLUDE for parameters whose values are arbitrary pieces of text. It is a different function only because it translates differently.

(TEXTAG TAG)

Quotes a text "tag", a place holder for a string of text which is part of the "value" of a TEXT-valued parameter; used within calls to TEXT. TAG should be in TEXTAGS, and should have a TRANS which is the string in question.

(TEXT LABEL PHRASE1 PHRASE2 ...)

Constructs a value for a TEXT-valued parameter; used in the value slot of calls to CONCLUDETEXT. The first argument is a label (or NIL) which may be used to tag the value for sorting by PRINTCONCLUSIONS. The remaining arguments are arbitrary forms to construct a text phrase. Result is a list (TEXT label . phrase). If there is only one argument, the label is interpreted as a text tag, i.e. (TEXT label) is the same as (TEXT label (TEXTAG label)).

I.6. Mapping functions

In complex and highly-developed knowledge bases, there may be rules that use parameters of contexts that are on different branches of the context tree. To access these parameters, it is necessary to use mapping functions. They exist to allow a rule to use parameters of the context type to which the rule is applied, as well as parameters of each of a specified list of context types.

All of the mapping functions have the arguments MAPSET, PRED, FREEVAR, and CARS. They map over the list MAPSET, performing some operation when PRED is true. MAPSET is usually a list of context instances, but it may be a list of the current values of a parameter. PRED is a predicate that has the same form as a rule premise. FREEVAR is the "iteration variable": on each iteration, it is set to the current element of MAPSET. It serves as the CNTXT variable in PRED—you can use it wherever you would ordinarily write CNTXT, to refer to the context element being mapped over. You can, of course, still use CNTXT inside PRED to refer to the context of the rule. FREEVAR is optional; if you leave that argument NIL, the default variable "FREEVAR" is used.

If MAPSET is a list of pairs (which it will be if the set is the result of a call to GETALL or GETOFFSPRING), the CARS argument to the mapping function should be set to T. This indicates that on each iteration, FREEVAR should be set to the CAR of the current element of MAPSET rather than to the element itself. This is always the case when using GETALL or GETOFFSPRING (below).

Most of the functions return ANSET set to their result. For THEREARE, whose result is a list, COLLECTEDLST is the default for ANSET; for those that return a single element, the default is FOUNDVAR. These result variables are global to the clause in which they are computed (but local to the rule); the result of a mapping function in a rule's PREMISE is often used in that rule's ACTION or in later PREMISE clauses.

If a mapping function is embedded in a call to the Lisp function NOT, the action and translation will be appropriately negated (i.e. the clause will be true if the function fails).

(THEREARE MAPSET PRED FREEVAR ANSET CARS DUPLES)

Collects all the elements of MAPSET for which PRED is true. If DUPLES is T, it returns a list of duples pairing each FREEVAR that succeeded with the value (a CF) returned when the predicate PRED was tested for that element; otherwise just returns a list of the successful freevars.

```
Ex: (THEREARE (GETALL KNOWNORG)
              ($AND (DEFINITE CNTXT IDENT))
              NIL COLLECTEDORGS T)
   -> You have examined the organisms isolated from positive cultures
      obtained from the patient, selecting those for which the
      identity of the organism is known with certainty.
```

[Sets COLLECTEDORGS to the resulting list.]

(THEREXISTS MAPSET PRED FREEVAR ANSET CARS)

Like THEREARE, but just finds the *first* freevar satisfying PRED, and returns that.

```
Ex: (THEREXISTS (GETALL CURTHER)
                ($AND (SAME FREEVAR DNAME (ONEOF AMPICILLIN
                                  CARBENICILLIN PENICILLIN METHICILLIN)))
                NIL NIL T)
   -> You have examined current drugs of the patient, and have found one
      for which the name of this drug is one of: ampicillin
      carbenicillin penicillin methicillin.
```

[This particular clause implicitly sets FOUNDVAR to result (a CURTHER context).]

(THEREARE! LST)

Returns true if LST is non-empty. This is like a call to THEREARE or THEREXISTS with PRED = T, but translates better, and sets no global variable. Used primarily for checking whether contexts of a certain type exist.

```
Ex: (THEREARE! (GETOFFSPRING CNTXT SMEARORG))
   -> There are organisms noted on smears of this culture.
```

(FORALL MAPSET PRED FREEVAR CARS)

True if PRED is true for each element of MAPSET. Function returns (trivially) true if map set is empty.

```
Ex:    (FORALL (GETALL POSCUL)
               ($AND (NOTSAME FREEVAR SPECSTAIN)
                     (NOTSAME FREEVAR CRYPTO-SEROLOGY)
                     (NOTSAME FREEVAR COCCI-SEROLOGY))
               NIL T)
   -> For each of the the positive cultures of the patient it is true
      that
        1) Organisms were not seen on the stain of this culture,
        2) The cryptococcal antigen in the csf was not positive, and
        3) The csf coccidioides serology was not positive
```

Note that, logically speaking, FORALL is the same as the clause:

(NOT (THEREXISTS MAPSET (NOT PRED) FREEVAR CARS)).

(FINDMAX MAPSET PRED TEST FREEVAR ANSET CARS)

On each iteration, evaluates TEST. Returns the FREEVAR for which TEST had the largest value (and PRED was true). Global MAXVAL is set to this maximum value of TEST, and may be used in the rule's ACTION or later in the PREMISE.

```
Ex:  (FINDMAX (GETALL CURTHER)
              ($AND (KNOWN $FREEDRUG WHENSTART))
              (VAL1 $FREEDRUG WHENSTART)
              $FREEDRUG NIL T)
   -> You have examined the current drugs of the patient for which the
      time since therapy with this drug was started is known,
      and have selected the one having the maximum value for the
      time since therapy with this drug was started.
```

(FINDMIN MAPSET PRED TEST FREEVAR ANSET CARS)

Like FINDMAX, but looks for the the smallest value of TEST, and sets global MINVAL.

Functions used within Mapping Functions

The functions VAL, GETALL, GETOFFSPRING, and APPEND are used as the MAPSET argument to mapping functions. NOTSAMEANS, SAMEANS, and TRACED are used as predicate clauses within the PRED argument (which is always a call to $AND). The functions VAL, GETALL, and GETOFFSPRING have an additional argument, notraceflg, that, if set, prevents the function from attempting to trace the parameter. This is used primarily by rules that want to inspect partial results and perhaps control the remainder of the tasks that the consultant is to perform.

(VAL CNTXT PARM)

Returns PARM of CNTXT as a list of (value CF) pairs, sorted by decreasing CF. VAL traces the parameter if it has not been traced yet.

(GETALL CTYPE NOTRACEFLG)

Returns a list of all contexts of type CTYPE in the form ((cntxt 1000)...). This is often used as the MAPSET argument.

(GETOFFSPRING CNTXT CTYPE NOTRACEFLG)

Like GETALL, but specialized to just one portion of the context tree. Returns a list of descendant contexts of CNTXT of type CTYPE in the form ((cntxt 1000) ...).

(APPEND LIST1 LIST2 ...)

Lisp function that appends its list arguments into one long list. It is used as the MAPSET argument when more than one type of context is to be examined.

(NOTSAMEANS CNTXT1 CNTXT2 PARM)

Premise clause that is true if CNTXT1 and CNTXT2 have different values for the parameter PARM. One of CNTXT1 and CNTXT2 is usually CNTXT, and the other the value of the FREEVAR arg.

(SAMEANS CNTXT1 CNTXT2 PARM)

A premise function that is true if CNTXT1 and CNTXT2 have the same value for the parameter PARM. One of CNTXT1 and CNTXT2 is usually CNTXT, and the other the value of the FREEVAR arg.

(TRACED CNTXT PARM)

True if PARM has been traced for CNTXT. This is used when the value of a parameter of one context is to be transferred to another context. To avoid circular reasoning, we specify that the target parameter must already be traced in order for a context to satisfy the predicate.

```
Ex: (THEREXISTS (APPEND (GETALL POSCUL) (GETALL PENDCUL))
                ($AND (TRACED FREEVAR NOSOCOMIAL)
                      (KNOWN FREEVAR NOSOCOMIAL)
                      (SAMEANS CNTXT FREEVAR SITE))
                NIL FOUNDCUL T)
   -> You have examined positive cultures obtained from the patient
      and pending cultures of the patient, and have found one for
      which
        1) All information about whether the infection was
           acquired while the patient was hospitalized has
           been gathered, and
        2) It is known whether the infection was acquired while
           the patient was hospitalized, and
        3) The culture under consideration and this culture have the
           same value for the site of the culture.
```

Conclusion Functions used in Rules with Mapping Functions

Sometimes a mapping function is used solely for its value as a predicate (e.g. FORALL, or THEREXISTS), or to compute a simple value to be used in the ACTION (e.g. the MAXVAL result of a FINDMAX might be used as the switch argument to CONCLUDET). In such cases, the normal conclusion functions are used, though possibly a variable from the premise will show up. However, more commonly a rule with mapping functions in it wants to manipulate the result of the map in its ACTION in other ways. The following functions are thus provided.

(CONCLIST CNTXT PARM GVAL TALLY)

GVAL is a list of duples (value CF). Concludes that PARM of CNTXT is each of those values, with the associated CF modified by TALLY.

```
Ex: (CONCLIST CNTXT IDENT GRIDVAL 900)
   -> There is strongly suggestive evidence (.9) that each of the ones
      that you found is the identity of the organism.
```

(CONCLUDEALL CNTXTS PARM VALU CF)

CNTXTS is a list of contexts, set by a mapping clause in the rule's PREMISE. Makes the same conclusion for each member of the list.

```
Ex: (CONCLUDEALL COLLECTEDCULS REQTHER YES -1000)
   -> It is definite (1.0) that the organisms isolated from the
      cultures that you selected should not be considered for therapy.
```

(TRANSPARM FROM TO PARM CF)

Transfers the value of PARM of FROM to TO, modified by CF. Both FROM and TO may be either a list of contexts or a single context. One of these is generally CNTXT, and the other the result of a mapping function in the premise (e.g., COLLECTEDLST or FOUNDVAR). If POSITIVE is set, only values with positive CFs are transferred.

```
Ex: (TRANSPARM FOUNDCUL CNTXT SECONDARY 1000)
   -> It is definite (1.0) that the information that you have gathered
      about the infection to which the bacteremia is secondary
      is also relevant to this culture.
```

(TRANSDIFPARM FROM FPARM TO TPARM CF POSITIVE)

Like TRANSPARM, but used when the values to be transferred are associated with two different parameters. FPARM and TPARM must, of course, have the same set of legal values (EXPECT

prop). Transfers the value of FPARM of FROM to TPARM of TO, modified by CF.

```
Ex: (TRANSDIFPARM COLLECTEDORGS IDENT CNTXT COVERFOR 700)
    -> There is suggestive evidence (.7) that the identity of each of
       the organisms that you selected is the organisms (other than
       those seen on cultures or smears) which might be causing
       the infection.
```

(TRANSLIST FROM TO PARMS CF)

Transfers to context TO the values of each parameter in PARMS of all contexts in FROM, modifying the CFs of each stored value by CF.

```
Ex: (TRANSLIST (VALYEW CNTXT SAMEBUG) CNTXT '(IDENT) 1000)
    -> It is definite (1.0) that these properties - ident - should be
       transferred from the organisms with possibly the same
       identity as this organism to this organism.
```

Appendix II
An Introduction to the Interlisp editor

The database editor checks most of the structures you enter for validity on some level (usually just syntactic, but the rule editor is more extensive), and may point out errors, attempt spelling correction, or ask you extra questions concerning what you just entered. Often when an error is detected, the program will call the Interlisp editor on the erroneous expression, asking you to fix it.

The editor prompts you with:

```
edit
*
```

often printing the expression as well. You should then give any edit commands you wish, finishing with OK. If you do not wish to edit the expression (i.e. you want to fix it some other way), or have reason to believe there really is no error, you can just type OK with no other commands to get out of the editor. If you change your mind in the middle of editing, the command STOP will exit and undo any changes you made (i.e. just as though you did no editing at all).[16]

We illustrate some of the commands with a detailed example, involving a previously entered CLOT rule.

```
Do you want to enter Rules, Parms, Go, etc (? for help) ? Rules

Rule# or NEW: 10
Translate, Delete, No change, or name of prop to modify: TRANSLATE

RULE010
-------
  If:  1) The ptt of the patient is normal,
       2) The pt of the patient is high,
       3) The tt of the patient is normal,
       4) The bleeding time of the patient is normal,
       5) The platelet count of the patient is normal, and
       6) The urea solubility test of the patient is normal
  Then:  It is definite (1.0) that the following is one of the coagulation
           defects of the patient: factor VII deficiency
  Author: Csd.bennett
  Literature: Harrisons
  Justification: Indication of extrinsic enzyme pathway defect
  Created:  2-Nov-80 17:42

Translate, Delete, No change, or name of prop to modify: P
RULE010
PREMISE: E   Typing E to this prompt puts us into the editor.
edit
($AND (SAME CNTXT PTT NORMAL) (SAME CNTXT PT HIGH) (SAME CNTXT TT NORMAL)
```

---

[16] Note: this is true only in calls to the Lisp editor from EMYCIN's editors. If you invoke the Lisp editor from Lisp yourself, STOP does *not* undo your changes

```
(SAME CNTXT BT NORMAL) (SAME CNTXT PC NORMAL) (SAME CNTXT FSF NORMAL))
```

Above is the Lisp form of the premise, and is what would be printed out if we were to give the command P. Because this expansion is so lengthy it is difficulty to read. There exists a command PP (for PrettyPrint), to print lists in a structured form.

```
*PP
($AND (SAME CNTXT PTT NORMAL)
      (SAME CNTXT PT HIGH)
      (SAME CNTXT TT NORMAL)
      (SAME CNTXT BT NORMAL)
      (SAME CNTXT PC NORMAL)
      (SAME CNTXT FSF NORMAL))
*3
```

A nonzero number is the command for moving to that subelement of the current expression. So the command 3 makes (SAME CNTXT PT HIGH) the object currently under consideration by the editor. In the above list, the atom $AND was the first element, the list (SAME CNTXT PTT NORMAL) was the second element. Commands of the form -n will make the current expression the n'th one from the end of the list

```
*P
(SAME CNTXT PT HIGH)
```

Here, the current element is short, we don't need to PP it; P suffices

```
*3
*P
PT
*0 P
(SAME CNTXT PT HIGH)
```

In the above line we gave two commands. The first command 0 (zero) makes the expression containing the current expression become the current expression. The command to print showed us that this had been accomplished, the current expression changed from being atom PT to the list containing it. Giving another 0 as shown below takes us up one level again.

```
*0 PP
($AND (SAME CNTXT PTT NORMAL)
      (SAME CNTXT PT HIGH)
      (SAME CNTXT TT NORMAL)
      (SAME CNTXT BT NORMAL)
      (SAME CNTXT PC NORMAL)
      (SAME CNTXT FSF NORMAL))

*2 P
(SAME CNTXT PTT NORMAL)
*(3 FSF) P
(SAME CNTXT FSF NORMAL)
```

Commands of the form (n s1 ... sM) replace the n'th element of the current expression with the expressions s1 ... sM.

```
*(4 HIGH) P
(SAME CNTXT FSF HIGH)
```

The other basic list modification commands are:

(n)	*deletes the n'th element of the current expression,*
(-n s1 ... sM)	*inserts s1 ... sM before the n'th element of the current expression,*
(N s1 ... sM)	*inserts s1 ... sM at the end of the current expression.*

```
*0 PP
($AND (SAME CNTXT FSF HIGH)
      (SAME CNTXT PT HIGH)
      (SAME CNTXT TT NORMAL)
      (SAME CNTXT BT NORMAL)
      (SAME CNTXT PC NORMAL)
      (SAME CNTXT FSF NORMAL))
```

The R (for Replace) command is illustrated below. (R s1 s2) will replace all the occurrences of s1 with s2. Note that both occurrences of FSF are replaced by PTT.

```
*R FSF PTT
*PP
($AND (SAME CNTXT PTT HIGH)
      (SAME CNTXT PT HIGH)
      (SAME CNTXT TT NORMAL)
      (SAME CNTXT BT NORMAL)
      (SAME CNTXT PC NORMAL)
      (SAME CNTXT PTT NORMAL))
```

Replace can be quite useful for correcting misspelled words, for example in the EXPECT's or TRANS's of parameters. We could have changed this rule's CF to .9 by editing the action and saying R 1000 900.

Shown below is the F (for Find) command. First we find the first occurrence of PTT, then after giving the command again we come to the second occurrence of PTT.

```
*F PTT
*P
... PTT HIGH)
```

The ... indicates the current expression begins in the middle of a list.

```
*F PTT
*P
... PTT NORMAL)
*(1 FSF)
```

This replacement of PTT with FSF restores the rule to its original form.

```
*0 0 PP
($AND (SAME CNTXT PTT HIGH)
      (SAME CNTXT PT HIGH)
      (SAME CNTXT TT NORMAL)
      (SAME CNTXT BT NORMAL)
```

```
      (SAME CNTXT PC NORMAL)
      (SAME CNTXT FSF NORMAL))
```

***MOVE 7 TO AFTER 2**

The Move command does just that: the seventh element in the list becomes the third.

```
*PP
($AND (SAME CNTXT PTT HIGH)
      (SAME CNTXT FSF NORMAL)
      (SAME CNTXT PT HIGH)
      (SAME CNTXT TT NORMAL)
      (SAME CNTXT BT NORMAL)
      (SAME CNTXT PC NORMAL))
*2
*(MBD ($OR *))
```

*(MBD s1 ... sM) replaces the current expression with s1 ... sM, replacing every occurrence of the atom * with the original current expression.*

***↑ PP**

The ↑ command returns us to the top of the expression being edited.

```
($AND ($OR (SAME CNTXT PTT HIGH))
      (SAME CNTXT FSF NORMAL)
      (SAME CNTXT PT HIGH)
      (SAME CNTXT TT NORMAL)
      (SAME CNTXT BT NORMAL)
      (SAME CNTXT PC NORMAL))
```

***MOVE 3 TO AFTER 2 2**

The third element in the list is moved to after the second element of the second element.

```
*PP
($AND ($OR (SAME CNTXT PTT HIGH)
           (SAME CNTXT FSF NORMAL))
      (SAME CNTXT PT HIGH)
      (SAME CNTXT TT NORMAL)
      (SAME CNTXT BT NORMAL)
      (SAME CNTXT PC NORMAL))
*(6)    We delete the sixth element of the list.
*OK     The command OK exits from the editor and returns us to EMYCIN.

Translate, Delete, No change, or name of prop to modify: TRANSLATE

RULE010
-------
  If:  1) A: The ptt of the patient is high, or
          B: The urea solubility test of the patient is normal,
       2) The pt of the patient is high,
       3) The tt of the patient is normal, and
       4) The bleeding time of the patient is normal
  Then:  It is definite (1.0) that the following is one of the coagulation
```

```
          defects of the patient: factor VII deficiency
Author: Csd.bennett
Literature: Harrisons
Justification: Indication of extrinsic enzyme pathway defect
Created:  2-Nov-80 17:42
```

Note: This rule no longer has any medical content and has been modified simply as a demonstration of the INTERLISP editor.

Listed below are the commands that have been already been introduced and some others which have not.

UNDO	Undoes the last change made to the list structure, and can be used repeatedly to undo several changes.
(B s1 ... sM)	inserts s1 ... sM before the current expression
(A s1 ... sM)	inserts s1 ... SM after the current expression
(: s1 ... sM)	replaces the current expression with s1 ... sM
(BI n m)	(for Both In) inserts a left parentheses before the n'th element and a right parentheses after the m'th element
(BO n)	removes the parentheses from around the n'th element
(n s1 ... sM)	replaces the n'th element of the current expression with the expressions s1 ... sM.
(n)	deletes the n'th element of the current expression
(-n s1 ... sM)	inserts s1 ... sM before the n'th element of the current expression
(N s1 ... sM)	attachs s1 ... sM to the end of the current expression
n	make the n'th element the current expression
0	make the list containing the current element the current element
↑	move to the top of the expression being edited
F atom	finds the first occurrence of atom, and makes it the current expression
MOVE @1 TO {BEFORE AFTER} @2	@1 specifies the location of the expression to be moved to before or after the expression located at @2
OK	return from the editor
!UNDO	undoes all the changes since you started editing this structure

Figure 9-2: Interlisp Editor Commands

References

[Bennett 78] Bennett, J.S., et al.
SACON: A knowledge-based consultant for structural analysis.
Technical Report, Computer Science Department, Stanford University, September, 1978.
Memo HPP-78-23.

[Bennett 80] Bennett, J.S., and Goldman, D.
CLOT: A knowledge-based consultant for bleeding disorders.
Technical Report, Computer Science Department, Stanford University, 1980.
Memo HPP-80-7.

[Bennett 81] Bennett, J.S., Hollander C.R.
DART: An Expert System for Computer Fault Diagnosis.
In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*. 1981.
Vancouver, B.C., Canada.

[Charniak 80] Charniak, E., Riesbeck, C.K., McDermott, D.V.
Artifical Intelligence Programming.
Lawrence Erlbaum Associates, 1980.

[Davis 76] Davis, R.
Applications of meta-level knowledge to the construction, maintenance, and use of large knowledge bases.
PhD thesis, Computer Science Department, Stanford University, June, 1976.

[Davis 77] Davis, R., Buchanan, B.G., and Shortliffe, E.H.
Production rules as a representation for a knowledge-based consultation system.
Artificial Intelligence 8(1):15-45, Feb, 1977.

[Fagan 79] Fagan, L.M.
Representation of dynamic clinical knowledge: measurement interpretation in the intensive care unit.
In *Proceedings of the Sixth International Joint Conference on Artificial Intelligence*, pages 260-262. 1979.
Tokyo, Japan.

[Heiser 78] Heiser, J.F., Brooks, R.E., Ballard, J.P.
Progress Report: A computerized psychopharmacology advisor.
In *Proceedings of the 11th Collegium Internationale Neuro-Psychopharmacologicum*. 1978.
Vienna.

[Horn 81] Horn, B.K.P., Winston, P.H.
LISP.
Addison Westley, 1981.

[Kunz 78] Kunz J.C., et al.
A physiological rule based system for interpreting pulmonary function test results.
Technical Report, Heuristic Programming Project, Computer Science Department, Stanford University, December, 1978.
HPP-78-19 Working Paper.

[Scott 77] Scott, A.C., et al.
Explanation capabilities of knowledge-based production systems.
American Journal of Computational Linguistics , 1977.
Microfiche 62.

[Shortliffe 75] Shortliffe E.H., and Buchanan B.G.
A model of inexact reasoning in medicine.
Mathematical Biosciences 23:351-379, 1975.

[Shortliffe 76] Shortliffe E.H.
Computer-based medical consultations: MYCIN.
American Elsevier, 1976.

[Teitelman 78] Teitelman.
Interlisp Reference Manual
XEROX Palo Alto Research Center, 1978.

[vanMelle 80] van Melle, W.
System aids in constructing consultation programs.
UMI Research Press, 1980.

[Yu 79] Yu, V.L., et al.
Evaluating the performance of a computer-based consultant.
Computer Programs in Biomedicine 9(1):95-102, Jan, 1979.

[Yu2 79] Yu, V.L., et al.
Antimicrobial selection by a computer: a blinded evaluation by infectious disease experts.
Journal of the American Medical Association 242(12):1279-1282, Sept, 1979.

Index

$AND 6, 39, 43, 52, 77, 118
$OR 52, 118

Action functions
  CONCLUDE 116
  CONCLUDE* 116
  CONCLUDET 116
  CONCLUDETEXT 121
  DO-ALL 117
  DONTASK 117
  NOTRELEVANT 117
  PRINTCONCLUSIONS 63, 118
  TEXT 121
  TEXTAG 121
ALIASES, global variable 101
ANY, value for EXPECT 32
ANY, value for LEGALVALS 33
ARL 10, 51
ARL, legal forms 54
ASK-ALL, value for MULTIVALUED 32
Auxiliary functions
  LISTOF 118
  MEASURE1 119
  ONEOF 118
  QUOTE 119
  SAMEP 77
  TRACED 123
  TRSEXP 114
  UNITS 115
  VAL 119
  VAL1 119
AUXPARMGROUPS 30, 37

Backward chaining 7, 8
Batch facility 12, 84

Case libraries 12, 71
Certainty factors (CF) 29, 5, 15, 43
CF, token for TEMPLATE 109
CFCOMBINE 44
CLOT 8, 20, 50
CNTXT, free variable 32, 35, 43, 52, 77
CNTXT, token for TEMPLATE 109
CNTXT, value for LEGALVALS 33
CONFIRM, used in CHECK 34
Context tree 7
  acquisition 21
  definition 15
  instance creation 42
  relationships 15, 19, 20
  uses of 18
Context types 4
  definition 15, 21
  examples 18, 19, 20
  root 17, 18, 23
CONTEXTTYPES, parameter group 22
Control structure 6
CRELST, global variable 102

DART 8, 34, 49
DATE, value for EXPECT 32
Debugging facilities 11
Defining Tables 95
DEL, parameter editor command 30
DO-ALL 40, 52
DOMAIN, global variable 101
DOMAIN.VARIABLES, an AUXPARMGROUP 37
DONTUNPACKCHARS, global variable 52
Dynamic Properties of Contexts 103
Dynamic Properties of Parameters 103

EDITTEXTFILE 71
Executive commands
  Build-system 64, 68
  Contexts 21, 64
  Debug/review 64
  Explanation 64
  Go 64
  Important-parms 64
  Knowledge-base-lister 64, 71, 105
  Lisp 64, 71
  Migrate-changes 64, 67
  New-consultation 64
  OK 64
  Parameters 64
  Quit 64
  Rules 64
  Save 64, 66, 69
  Typescript 64
  Variables 29, 64
  Words 64, 105
Explanation, WHY and HOW 9, 75

Fact triples 3, 4, 6, 8, 15
Files
BATCH.NUMBERS 85
CHANGES 66, 69
hash files 70
help file 71
knowledge base 66
knowledge-base file extension 66
migration 67
QUEST 104
rules and cases available 67
FINDOUT, the rule interpreter 41
FIXP, value for EXPECT 32, 33, 34
Functions 111
auxiliary 118
Conclusion 115
for TEXT-valued parameters 119
mapping 121
non-numeric predicates 111
numeric predicates 114
user-defined 107
FUNCTIONS, an AUXPARMGROUP 37, 109

GETPARMS 30
Goals and subgoals 4, 6, 7, 15, 18, 27, 55
GRAVIDA 8, 20
GROUP, parameter editor command 30, 45
GUESTS, global variable 69

HACKERS, global variable 69
HEADMED 8
Human-engineering facilities 12

IGNORE, token for TEMPLATE 109
IMPORTANTPARMS, global variable 77, 12, 84
Inference structure 55
Interlisp 1, 3, 102
Interiisp editor 1, 127

Knowledge acquisition facilities 5, 10
Knowledge base listing 71

Lisp 1, 6, 10, 35, 43, 65, 69, 77, 108
LITHO 8, 19

MAINDIRECTORY, global variable 67, 69
MAKELIBRARY 71
Mapping functions
APPEND 123
CONCLIST 124
CONCLUDEALL 124
FINDMAX 122
FINDMIN 123
FORALL 122
GETALL 123
GETOFFSPRING 123
NOTSAMEANS 123
SAMEANS 123
THEREARE 121
THEREARE! 122
THEREXISTS 122
TRANSDIFPARM 124
TRANSLIST 125
TRANSPARM 124
VAL 123
MYCIN 3, 5, 7, 17, 18, 22, 26, 29, 36, 46, 49, 75, 81, 93, 97, 98
MYCINHASHFILES, global variable 66

NAME, parameter editor command 30
NUMB, value for EXPECT 32, 33, 34

Options
BATCH 84
CHANGE question answer 90
consultation tracing 11
during consultation 90
during tabular input 94
EX 75
EXPLAIN 75, 77, 78, 90
FORGET 90
FT (TRACE) 75
NOFIX 89
NOPR 89
RENAME 92
REVIEW 11, 75, 77, 80
SAVE 92
Special Options 89
SPECIAL.OPTIONS, global variable 84
SUMMARY 26, 92, 97
TAB 26, 93
TER 89
TEST 75, 77, 79
TRACE 75
TS 89
UPDATE 35, 92
WHY and HOW 75

Parameter editor commands 30
Parameter group 27, 44

Parameters 4, 27
definition 30
expected values 28
multiple-valued 29
screening 35
single-valued 29
yes/no 29
PARM, free variable 35
PARM, token for TEMPLATE 109
PARMGROUPS 30
PARMS, token for TEMPLATE 109
PDTCHANGEPROPS, global variable 84, 86
POSNUMB, value for EXPECT 32, 33, 34
Predicate functions
BETWEEN* 115
DEFINITE 113
DEFIS 112
DEFNOT 113
GREATEQ* 115
GREATERP* 115
KNOWN 113
LESSEQ* 115
LESSP* 115
MIGHTBE 112
NOTDEFINITE 113
NOTDEFIS 113
NOTDEFNOT 113
NOTKNOWN 113
NOTSAME 112
ONCEKNOWN 113
ONCEKNOWN* 114
SAME 112, 120
THOUGHTNOT 112
VNOTKNOWN 112
Preview, of rules 42
PRINTIDS, global variable 101
Properties
ACTION 39
ALWAYSEVAL 43
ANTECEDENT 41
ANTECEDENT-IN 35, 102
APPLIEDTO 104
ASKED 103
ASKFIRST 23, 25, 33, 42, 63
ASSOCWITH 26
AUTHOR 41
CASES 41
CHECK 34
CNTXTSUMMARY 26, 92, 97
COMMENTS 41
COMPOUND 106
CONCEPT 107
CONCLUSIONS 103
CONTAINED-IN 35, 102
CONTEXT 45
CRE8 103
CTRANS 46
DATE 41
EXPECT 29, 30, 31, 34
EXPECTED 107
GOALS 21, 22, 23, 24, 41, 50, 56, 63
HYPART 107
INCONCEPT 107
INFUNCS 107
INITIALDATA 21, 22, 23, 24, 41, 50, 56, 63, 93
INTERNAL 107
JUSTIFICATION 41
LABEL.ORDER 35
LASTEDIT 41
LEGALTERM 26, 96
LEGALVALS 33, 35
LITERATURE 41
MULTIVALUED 30, 32
MUSTMATCH 107
NOCONTENT 106
OFFSPRING 26
PARMGROUP 17, 22, 23
PARMSOF 103
PATWORD 107
PRECEDENCE 110
PREMISE 39
PRINTID 22, 23, 102
PROMPT 12, 29, 31, 34, 42, 63
PROMPT1ST 24, 43
PROMPT2ND 24, 43, 93
PROMPTEVER 24, 43
PROPERNOUN 33
RANGE 34
REPROMPT 13, 29, 31
RULE.OUT 107
RULETYPES 17, 22, 26, 46
SPEC 107
SPECIAL 34
SUBJECT 17, 42, 44, 101
SVAL 46
SYN 25
TABHEAD 26, 96
TABPARMS 26, 96
TABPROMPT 96
TABSTOPS 26, 96

TEMPLATE 109
TERSEHEAD 26, 96
TRACED 103
TRANS 22, 23, 31, 96, 109
TRANS for functions 109
TXTSYN 106
TXTSYNOF 107
UNIQUE 25
UNITS 33, 52
UPDATE 35, 92
UPDATED-BY 35, 102
UPDATED-BY-THE-WAY 35, 102
UPDATED-IN 35, 102
UPPOINT 103
USED-BY 35, 102
USED-BY-THE-WAY 35, 102
VALUESYN 106
VALUESYNOF 107
XTRASPECIAL 34
PUFF 8, 17

QA
Dictionary 9, 67, 81, 105
legal question forms 83
modifying 104
Module 9, 12, 75, 79, 81
variables and functions 104
QA options
ADVICE 82
EQ 82
LP 82
PR 82
QT 82

ROOTID, global variable 101
ROOTNAMES, global variable 101
ROOTNODE, global variable 102
ROOTPARM, global variable 101
Rule evaluation 43
Rule Groups 44
RULEGROUPS, an AUXPARMGROUP 37, 45
Rules
actions 6
antecedent 46, 63
checking 10
consequent 46
evaluation 6
goal 50, 56, 64, 95
interpreter 6, 41
pointers 101
predicates 6
premises 6
Preview 42
production 5, 6, 7, 15
self-referencing 42, 49
Unitypath 42

SACON 3, 5, 6, 7, 15, 19, 27, 47, 55, 81, 108
SREFMARK, in knowledge-base listings 74
SYSFUNCTIONS, global variable 109
SYSTEMNAME, global variable 68

TABLE, token for TEMPLATE 109
TABLES, an AUXPARMGROUP 37, 93
Tabular input facility 93
TALLY, free variable 43
TALLY, global variable 43
TALLY, token for TEMPLATE 109
Teiresias 11
TEXT, token for TEMPLATE 109
TEXT, value for EXPECT 33
TEXT, value for LEGALVALS 35
TEXTAGS, an AUXPARMGROUP 37
Tracing, parameters 6, 41
Translation
context types 25
English 6, 9, 10
terse 40
TRANSLATOR, global variable 101
TRANSVERBS, global variable 31
TREEROOT, global variable 101
TTYIN, input facility 13

Unitypath, of rules 42

VALU, free variable 31, 34, 35
VALU, token for TEMPLATE 109
Values of parameters 4, 29
VALUS, token for TEMPLATE 109
VM 9